

Preface

An Innovative C Programming Textbook for the 2020s

Good programmers write code that humans can understand. 1

-Martin Fowler

I think that it's extraordinarily important that we in computer science keep fun in computing.²

—Alan Perlis

Welcome to *C How to Program, Ninth Edition*. We present a friendly, contemporary, code-intensive, case-study-oriented introduction to C—which is among the world's most popular programming languages.³ Whether you're a student, an instructor or a professional programmer, this book has much to offer you. In this Preface, we present the "soul of the book."

At the heart of the book is the Deitel signature live-code approach—we generally present concepts in the context of 147 complete, working, real-world C programs, rather than in code snippets. We follow each code example with one or more live program input/output dialogs. All the code is provided free for download at

```
https://deitel.com/c-how-to-program-9-e
https://pearson.com/deitel
```

You should execute each program in parallel with reading the text, making your learning experience "come alive."

For many decades:

- computer hardware has rapidly been getting faster, cheaper and smaller,
- Internet bandwidth (that is, its information-carrying capacity) has rapidly been getting larger and cheaper, and
- quality computer software has become ever more abundant and often free or nearly free through the open-source movement.

^{1.} Martin Fowler (with contributions by Kent Beck). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. p. 15.

^{2.} Alan Perlis, Quoted in the book dedication of *The Structure and Interpretation of Computer Programs, 2/e* by Hal Abelson, Gerald Jay Sussman and Julie Sussman. McGraw-Hill. 1996.

^{3.} Tiobe Index for November 2020. Accessed November 9, 2020. https://www.tiobe.com/tiobe-index/.

We'll say lots more about these important trends. The Internet of Things (IoT) is already connecting tens of billions of computerized devices of every imaginable type. These generate enormous volumes of data (one form of "big data") at rapidly increasing speeds and quantities. And most computing will eventually be performed online in "the Cloud"—that is, by using computing services accessible over the Internet.

For the novice, the book's early chapters establish a solid foundation in programming fundamentals. The mid-range to high-end chapters and the 20+ case studies ease novices into the world of professional software-development challenges and practices.

Given the extraordinary performance demands that today's applications place on computer hardware, software and the Internet, professionals often choose C to build the most performance-intensive portions of these applications. Throughout the book, we emphasize performance issues to help you prepare for industry.

The book's modular architecture (see the chart on the inside front cover) makes it appropriate for several audiences:

- Introductory and intermediate college programming courses in Computer Science, Computer Engineering, Information Systems, Information Technology, Software Engineering and related disciplines.
- Science, technology, engineering and math (STEM) college courses with a programming component.
- Professional industry training courses.
- Experienced professionals learning C to prepare for upcoming softwaredevelopment projects.

We've enjoyed writing nine editions of this book over the last 29 years. We hope you'll find C How to Program, 9/e informative, challenging and entertaining as you prepare to develop leading-edge, high-performance applications and systems in your career.

New and Updated Features in This Ninth Edition

Here, we briefly overview some of this edition's new and updated features. There are many more. The sections that follow provide more details:

- We added a **one-page color Table of Contents chart** on the inside front cover, making it easy for you to see the entire book from "40,000 feet." This chart emphasizes the book's **modular architecture** and lists most of the case studies.
- Some of the case studies are book sections that walk through the complete source code—these are supported by end-of-chapter exercises that might ask you to modify the code presented in the text or take on related challenges. Some are exercises with detailed specifications from which you should be able to develop the code solution on your own. Some are exercises that ask you to visit websites that contain nice tutorials. And some are exercises that ask you to visit developer websites where there may be code to study, but no tutorials—and the code may not be well commented. Instructors will decide which of the case studies are appropriate for their particular audiences.

- We adhere to the C11/C18 standards.
- We tested all the code for correctness on the Windows, macOS and Linux operating systems using the latest versions of the Visual C++, Xcode and GNU gcc compilers, respectively, noting differences among the platforms. See the **Before You Begin** section that follows this Preface for software installation instructions.
- We used the clang-tidy static code analysis tool to check all the code in the book's code examples for improvement suggestions, from simple items like ensuring variables are initialized to warnings about potential security flaws. We also ran this tool on the code solutions that we make available to instructors for hundreds of the book's exercises. The complete list of code checks can be found at https://clang.llvm.org/extra/clang-tidy/checks/list.html.
- GNU gcc tends to be the most compliant C compiler. To enable macOS and Windows users to use gcc if they wish, Chapter 1 includes a test-drive demonstrating how to compile programs and run them using gcc in the cross-platform GNU Compiler Collection Docker container.
- We've added 350+ integrated Self-Check exercises, each followed immediately by its answer. These are ideal for self study and for use in "flipped classrooms" (see the "Flipped Classrooms" section later in this Preface).
- To ensure that book content is **topical**, we did extensive Internet research on C specifically and the world of computing in general, which influenced our choice of case studies. We show C as it's intended to be used with a rich collection of applications programming and systems programming case studies, focusing on computer-science, artificial intelligence, data science and other fields. See the "Case Studies" section later in this Preface for more details.
- In the text, code examples, exercises and case studies, we familiarize students with current topics of interest to developers, including open-source software, virtualization, simulation, web services, multithreading, multicore hardware architecture, systems programming, game programming, animation, visualization, 2D and 3D graphics, artificial intelligence, natural language processing, machine learning, robotics, data science, secure programming, cryptography, Docker, GitHub, StackOverflow, forums and more.
- We adhere to the latest ACM/IEEE computing curricula recommendations, which call for covering security, data science, ethics, privacy and performance concepts and using real-world data throughout the curriculum. See the "Computing and Data Science Curricula" section for more details.
- Most chapters in this book's recent editions end with Secure C programming sections that focus on the SEI CERT C Coding Standard from the CERT group of Carnegie Mellon University's Software Engineering Institute (SEI). For this edition, we tuned the SEI CERT-based sections. We also added security icons in the page margin whenever we discuss a security-related issue in the text. All of this is consistent with the ACM/IEEE computing curricula docu-



- ments' enhanced emphasis on security. See the "Computing and Data Science Curricula" section later in this Preface for a list of the key curricula documents.
- Consistent with our richer treatment of security, we've added case studies on secret-key and public-key cryptography. The latter includes a detailed walkthrough of the enormously popular RSA algorithm's steps, providing hints to help you build a working, simple, small-scale implementation.
- We've enhanced existing case studies and added new ones focusing on AI and data science, including simulations with random-number generation, survey data analysis, natural language processing (NLP) and artificial intelligence (machine-learning with simple linear regression). Data science is emphasized in the latest ACM/IEEE computing curricula documents.
- We've added exercises in which students use the Internet to research ethics and privacy issues in computing.
- We tuned our mutltithreading and multicore performance case study. We also show a performance icon in the margin whenever we discuss a performance-related issue in the text.
- We integrated the previous edition's hundreds of software-development tips directly into the text for a smoother reading experience. We call out **common errors** and **good software engineering practices** with new margin icons.
- We upgraded our appendix on additional sorting algorithms and analysis of algorithms with Big O to full-chapter status (Chapter 13).
- C programmers often subsequently learn one or more C-based object-oriented languages. We added an appendix that presents a friendly intro to object-oriented programming concepts and terminology. C is a procedural programming language, so this appendix will help students appreciate differences in thinking between C developers and the folks who program in languages like C++, Java, C#, Objective-C, Swift and other object-oriented languages. We do lots of things like this in the book to prepare students for industry.
- Several case studies now have you use free open-source libraries and tools.
- We added a case study that performs visualization with gnuplot.
- We removed the previous edition's introduction to C++ programming to make room for the hundreds of integrated self-check exercises and our new applications programming and systems programming case studies.
- This new edition is published in a larger font size and page size for enhanced readability.

A Tour of the Book

The **Table of Contents** graphic on the inside front cover shows the book's **modular architecture**. Instructors can conveniently adapt the content to a variety of courses and audiences. Here we present a brief chapter-by-chapter walkthrough and indicate where





the book's case studies are located. Some are in-chapter examples and some are end-of-chapter exercises. Some are fully coded. For others, you'll develop the solution.

Chapters 1–5 are traditional introductory C programming topics. Chapters 6–11 are intermediate topics forming the high end of Computer Science 1 and related courses. Chapters 12–15 are advanced topics for late CS1 or early CS2 courses. Here's a list of the topical, challenging and often entertaining hands-on case studies.

Systems Programming Case Studies

- Systems Software—Building Your Own Computer (as a virtual machine)
- Systems Software—Building Your Own Compiler
- Embedded Systems Programming—Robotics, 3D graphics and animation with the Webots Simulator
- Performance with Multithreading and Multicore Systems

Application Programming Case Studies

- Algorithm Development—Counter-Controlled Iteration
- Algorithm Development—Sentinel-Controlled Iteration
- Algorithm Development—Nested Control Statements
- Random-Number Simulation—Building a Casino Game
- Random-Number Simulation—Card Shuffling and Dealing
- Random-Number Simulation—The Tortoise and the Hare Race
- Intro to Data Science—Survey Data Analysis
- Direct-Access File Processing—Building a Transaction-Processing System
- Visualizing Searching and Sorting Algorithms—Binary Search and Merge Sort.
- Artificial Intelligence/Data Science—Natural Language Processing ("Who Really Wrote the Works of William Shakespeare?")
- Artificial Intelligence/Data Science—Machine Learning with the GNU Scientific Library ("Statistics Can Be Deceiving" and "Have Average January Temperatures in New York City Been Rising Over the Last Century?")
- Game Programming—Cannon Game with the raylib Library
- Game Programming—SpotOn Game with the raylib Library
- Multimedia: Audio and Animation—The Tortoise and the Hare Race with the raylib Library
- Security and Cryptography—Implementing a Vigenère Secret-Key Cipher and RSA Public-Key Cryptography
- Animated Visualization with raylib—The Law of Large Numbers
- Web Services and the Cloud—Getting a Weather Report Using libcurl and the OpenWeatherMap Web Services, and An Introduction to Building Mashups with Web Services.

xxiv Preface

Whether you're a student getting a sense of the textbook you'll be using, an instructor planning your course syllabus or a professional software developer deciding which chapters to read as you prepare for a project, the following chapter overviews will help you make the best decisions.

Part 1: Programming Fundamentals Quickstart

Chapter 1, Introduction to Computers and C, engages novice students with intriguing facts and figures to excite them about studying computers and computer programming. The chapter includes current technology trends, hardware and software concepts and the data hierarchy from bits to databases. It lays the groundwork for the C programming discussions in Chapters 2–15, the appendices and the integrated case studies.

We discuss the programming-language types and various technologies you're likely to use as you develop software. We introduce the C standard library—existing, reusable, top-quality, high-performance functions to help you avoid "reinventing the wheel." You'll enhance your productivity by using libraries to perform significant tasks while writing only modest numbers of instructions. We also introduce the **Internet**, the **World Wide Web**, the "Cloud" and the **Internet of Things** (**IoT**), laying the groundwork for modern applications development.

This chapter's test-drives demonstrate how to compile and execute C code with

- Microsoft's Visual C++ in Visual Studio on Windows,
- Apple's Xcode on macOS, and
- GNU's gcc on Linux.

We've run the book's 147 code examples using each environment.⁴ Choose whichever program-development environment you prefer—the book works well with others, too.

We also demonstrate GNU gcc in the GNU Compiler Collection Docker container. This enables you to run the latest GNU gcc compiler on Windows, macOS or Linux—this is important because the GNU compilers generally implement all (or most) features in the latest language standards. See the Before You Begin section that follows this Preface for compiler installation instructions. See the Docker section later in this Preface for more information on this important developer tool. For Windows users, we point to Microsoft's step-by-step instructions that allow you to install Linux in Windows via the Windows Subsystem for Linux (WSL). This is another way to be able to use the GNU gcc compiler on Windows.

You'll learn just how big "big data" is and how quickly it's getting even bigger. The chapter closes with an introduction to artificial intelligence (AI)—a key overlap between the computer-science and data-science fields. AI and data science are likely to play significant roles in your computing career.

Chapter 2, Intro to C Programming, presents C fundamentals and illustrates key language features, including input, output, fundamental data types, computer memory concepts, arithmetic operators and their precedence, and decision making.

^{4.} We point out the few cases in which a compiler does not support a particular feature.

Chapter 3, Structured Program Development, is one of the most important chapters for programming novices. It focuses on problem-solving and algorithm development with C's control statements. You'll develop algorithms through top-down, stepwise refinement, using the if and if...else selection statements, the while iteration statement for counter-controlled and sentinel-controlled iteration, and the increment, decrement and assignment operators. The chapter presents three algorithm-development case studies.

Chapter 4, Program Control, presents C's other control statements—for, do...while, switch, break and continue—and the logical operators. A key feature of this chapter is its structured-programming summary.

Chapter 5, Functions, introduces program construction using existing and custom functions as building blocks. We demonstrate simulation techniques with random-number generation. We also discuss passing information between functions and how the function-call stack and stack frames support the function call/return mechanism. We begin our treatment of recursion. This chapter also presents our first simulation case study—Building a Casino Game, which is enhanced by end-of-chapter exercises.

Part 2: Arrays, Pointers and Strings

Chapter 6, Arrays, presents C's built-in array data structure for representing lists and tables of values. You'll define and initialize arrays, and refer to their individual elements. We discuss passing arrays to functions, sorting and searching arrays, manipulating multidimensional arrays and creating variable-length arrays whose size is determined at execution time. Chapter 13, Computer-Science Thinking: Sorting Algorithms and Big O, discusses more sophisticated and higher-performance sorting algorithms and presents a friendly introduction to analysis of algorithms with computer science's Big O notation. Chapter 6 presents our first data-science case study—Intro to Data Science: Survey Data Analysis. In the exercises, we also present two Game Programming with Graphics, Sound and Collision Detection case studies and an Embedded Systems Programming case study (Robotics with the Webots Simulator).

Chapter 7, Pointers, presents what is arguably C's most powerful feature. Pointers enable programs to

- accomplish pass-by-reference,
- pass functions to other functions, and
- create and manipulate dynamic data structures, which you'll study in detail in Chapter 12.

The chapter explains pointer concepts, such as declaring pointers, initializing pointers, getting the memory address of a variable, dereferencing pointers, pointer arithmetic and the close relationship between arrays and pointers. This chapter presents our first systems software case-study exercise—Building Your Own Computer with Simulation. This case study introduces an essential modern computer-architecture topic—virtual machines.

xxvi Preface

Chapter 8, Characters and Strings, introduces the C standard library's string, character and memory-block processing functions. You'll use these powerful capabilities in Chapter 11, File Processing, as you work through a natural language processing (NLP) case study. You'll see that strings are intimately related to pointers and arrays.

Part 3: Formatted Input/Output, Structures and File Processing

Chapter 9, Formatted Input/Output, discusses the powerful formatting features of functions scanf and printf. When properly used, these functions securely input data from the standard input stream and output data to the standard output stream, respectively.

Chapter 10, Structures, Unions, Bit Manipulation and Enumerations, introduces structures (structs) for aggregating related data items into custom types, unions for sharing memory among multiple variables, typedefs for creating aliases for previously defined data types, bitwise operators for manipulating the individual bits of integral operands and enumerations for defining sets of named integer constants. Many C programmers go on to study C++ and object-oriented programming. In C++, C's structs evolve into classes, which are the "blueprints" C++ programmers use to create objects. C structs contain only data. C++ classes can contain data *and* functions.

Chapter 11, File Processing, introduces files for long-term data retention, even when the computer is powered off. Such data is said to be "persistent." The chapter explains how plain-text files and binary files are created, updated and processed. We consider both sequential-access and random-access file processing. In one of our case-study exercises, you'll read data from a comma-separated value (CSV) file. CSV is one of the most popular file formats in the data-science community. This chapter presents our next case study—Building a Random-Access Transaction-Processing System. We use random-access files to simulate the kind of high-speed direct-access capabilities that industrial-strength database-management systems have. This chapter also presents our first artificial-intelligence/data-science case study, which uses Natural Language **Processing (NLP)** techniques to begin investigating the controversial question, "Who really wrote the works of William Shakespeare?" A second artificial-intelligence/datascience case study—Machine Learning with the GNU Scientific Library—investigates Anscombe's Quartet using simple linear regression. This is a collection of four dramatically different datasets that have identical or nearly identical basic descriptive statistics. It offers a valuable insight for students and developers learning some datascience basics in this computer-science textbook. The case study then asks you to run a simple linear regression on 126 years of New York City average January temperature data to determine if there is a cooling or warming trend.

^{5. &}quot;Anscombe's Quartet." Accessed November 13, 2020. https://en.wikipedia.org/wiki/Anscombe%27s_quartet.

Part 4: Algorithms and Data Structures

Chapter 12, Data Structures, uses structs to aggregate related data items into custom types, typedefs to create aliases for previously defined types, and dynamically linked data structures that can grow and shrink at execution time—linked lists, stacks, queues and binary trees. You can use the techniques you learn to implement other data structures. This chapter also presents our next systems-software case study exercise—Building Your Own Compiler. We'll define a simple yet powerful highlevel language. You'll write some high-level-language programs that your compiler will compile into the machine language of the computer you built in Chapter 7. The compiler will place its machine-language output into a file. Your computer will load the machine language from the file into its memory, execute it and produce appropriate outputs.

Chapter 13, Computer-Science Thinking: Sorting Algorithms and Big O, introduces some classic computer-science topics. We consider several algorithms and compare their processor demands and memory consumption. We present a friendly introduction to computer science's Big O notation, which indicates how hard an algorithm may have to work to solve a problem, based on the number of items it must process. The chapter includes the case study Visualizing the High-Performance Merge Sort.

Our recursion (Chapter 5), arrays (Chapter 6), searching (Chapter 6), data structures (Chapter 12), sorting (Chapter 13) and Big O (Chapter 13) coverage provides nice content for a C data structures course.

Part 5: Preprocessor and Other Topics

Chapter 14, Preprocessor, discusses additional features of the C preprocessor, such as using #include to help manage files in large programs, using #define to create macros with and without arguments, using conditional compilation to specify portions of a program that should not always be compiled (e.g., extra code used only during program development), displaying error messages during conditional compilation, and using assertions to test whether expressions' values are correct.

Chapter 15, Other Topics, covers additional C topics, including multithreading support (available in GNU gcc, but not Xcode or Visual C++), variable-length argument lists, command-line arguments, compiling multiple-source-file programs, extern declarations for global variables in other files, function prototypes, restricting scope with static, makefiles, program termination with exit and atexit, suffixes for integer and floating-point literals, signal handling, dynamic memory-allocation functions calloc and realloc and unconditional branching with goto. This chapter presents our final case study—Performance with Multithreading and Multicore Systems. This case study demonstrates how to create multithreaded programs that will run faster (and often much faster) on today's multicore computer architectures. This is a nice capstone

xxviii Preface

case study for a book about C, for which writing high-performance programs is paramount.

Appendices

Appendix A, Operator Precedence Chart, lists C's operators in highest-to-lowest precedence order.

Appendix B, ASCII Character Set, shows characters and their corresponding numeric codes.

Appendix C, Multithreading/Multicore and Other C18/C11/C99 Topics, covers designated initializers, compound literals, type bool, complex numbers, additions to the preprocessor, the restrict keyword, reliable integer division, flexible array members, relaxed constraints on aggregate initialization, type generic math, inline functions, return without expression, __func__ predefined identifier, va_copy macro, C11 headers, _Generic keyword (type generic expressions), quick_exit function, Unicode® support, _noreturn function specifier, type-generic expressions, Annex L: Analyzability and Undefined Behavior, memory-alignment control, static assertions, floating-point types and the timespec_get function.

Appendix D, Intro to Object-Oriented Programming Concepts, presents a friendly overview of object-oriented programming terminology and concepts. After learning C, you'll likely also learn one or more C-based object-oriented languages—such as C++, Java, C#, Objective-C or Swift—and use them side-by-side with C.

Online Appendices

Appendix E, Number Systems, introduces the binary, octal, decimal and hexadecimal number systems.

Appendices F–H, Using the Visual Studio Debugger, Using the GNU gdb Debugger and Using the Xcode Debugger, demonstrate how to use our three preferred compilers' basic debugging capabilities to locate and correct execution-time problems in your programs.

C How to Program, 9/e Key Features

C Programming Fundamentals

In our rich coverage of C fundamentals:

- We emphasize problem-solving and algorithm development.
- To help students prepare to work in industry, we use the terminology from the latest **C** standard documents in preference to general programming terms.
- We avoid heavy math, leaving it to upper-level courses. Optional mathematical exercises are included for science and engineering courses.

CII and CI8 Standards

C11 refined and expanded C's capabilities. We've added more features from the C11 standard. Since C11, there has been only one new version, C18.⁶ It "addressed defects in C11 without introducing new language features."⁷

Innovation: "Intro-to" Pedagogy with 350+ Integrated Self-Check Exercises

This book uses our new "Intro to" pedagogy with integrated Self Checks and answers. We introduced this pedagogy in our recent textbook, Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud.

- Chapter sections are intentionally small. We use a "read-a-little, do-a-little, test-a-little" approach. You read about a new concept, study and execute the corresponding code examples, then test your understanding of the new concept via the integrated Self-Check exercises immediately followed by their answers. This will help you keep a brisk learning pace.
- Fill-in-the-blank, true/false and discussion Self Checks enable you to test your understanding of the concepts and terminology you've just studied.
- Code-based Self Checks give you a chance to use the terminology and reinforce the programming techniques you've just studied.
- The **Self-Checks** are particularly valuable for **flipped classroom** courses—we'll soon say more about that popular educational phenomenon.

KIS (Keep It Simple), KIS (Keep it Small), KIT (Keep it Topical)

- Keep it simple—We strive for simplicity and clarity.
- **Keep it small**—Many of the book's examples are small. We use more substantial code examples, exercises and projects when appropriate, particularly in the case studies that are a core feature of this textbook.
- Keep it topical—"Who dares to teach must never cease to learn." (J. C. Dana)—In our research, we browsed, read or watched thousands of current articles, research papers, white papers, books, videos, webinars, blog posts, forum posts, documentation pieces and more.

^{6.} ISO/IEC 9899:2018, Information technology — Programming languages — C, https://www.iso.org/standard/74528.html.

^{7.} https://en.wikipedia.org/wiki/C18_(C_standard_revision). Also http://www.iso-9899.info/wiki/The_Standard.

^{8.} John Cotton Dana. From https://www.bartleby.com/73/1799.html: "In 1912 Dana, a Newark, New Jersey, librarian, was asked to supply a Latin quotation suitable for inscription on a new building at Newark State College (now Kean University), Union, New Jersey. Unable to find an appropriate quotation, Dana composed what became the college motto.—*The New York Times Book Review*, March 5, 1967, p. 55."

Hundreds of Contemporary Examples, Exercises and Projects (EEPs)

You'll use a hands-on applied approach to learn from a broad selection of real-world examples, exercises and projects (EEPs) drawn from computer science, data science and other fields:

- You'll attack exciting and entertaining challenges in our larger case studies, such as building a casino game, building a survey-data-analysis program, building a transaction-processing system, building your own computer (using simulation to build a virtual machine), using AI/data-science technologies such as natural language processing and machine learning, building your own compiler, programming computer games, programming robotics simulations with Webots, and writing multithreaded code to take advantage of today's multicore computer architectures to get the best performance from your computer.
- Research and project exercises encourage you to go deeper into what you've learned and explore other technologies. We encourage you to use computers and the Internet to solve significant problems. Projects are often more extensive in scope than the exercises—some might require days or weeks of implementation effort. Many of these are appropriate for class projects, term projects, directed-study projects, capstone-course projects and thesis research. We do not provide solutions to the projects.
- Instructors can tailor their courses to their audience's unique requirements and vary labs and exam questions each semester.

Working with Open-Source Software

In those days [batch processing] programmers never even documented their programs, because it was assumed that nobody else would ever use them. Now, however, time-sharing had made exchanging software trivial: you just stored one copy in the public repository and thereby effectively gave it to the world. Immediately people began to document their programs and to think of them as being usable by others. They started to build on each other's work. —Robert Fano, Founding Director of MIT's Project MAC in the 1960s, which evolved into today's Computer Science and Artificial Intelligence Laboratory (CSAIL)¹⁰

Open source is software with source code that anyone can inspect, modify, and enhance." We encourage you to try lots of **demos** and view free, **open-source** code examples (available on sites such as **GitHub**) for inspiration. We say more about GitHub in the section "Thinking Like a Developer—GitHub, StackOverflow and More."

^{9.} Robert Fano, quoted in *Dream Machine: J.C.R. Licklider and the Revolution That Made Computing Personal* by Mitchell Waldrop. Penguin Putnam, 2002. p. 232.

^{10. &}quot;MIT Computer Science and Artificial Intelligence Laboratory." Accessed November 9, 2020. https://en.wikipedia.org/wiki/MIT_Computer_Science_and_Artificial_Intelligence_Laboratory.

^{11. &}quot;What is open source?" Accessed November 14, 2020. https://opensource.com/resources/what-open-source.

Visualizations

We include high-level **visualizations** produced with the **gnuplot** open-source visualization package to reinforce your understanding of the concepts:

- We use visualizations as a pedagogic tool. For instance, one example makes the law of large numbers "come alive" in a dice-rolling simulation (see Chapter 10—Raylib Game Programming Case Studies later in this Preface). As this program performs increasing numbers of die rolls, you'll see each of the six faces' (1, 2, 3, 4, 5, 6) percentage of the total rolls gradually approach 16.667% (1/6th), and the lengths of the bars representing the percentages equalize.
- You should experiment with the code to implement your own visualizations.

Data Experiences

In the book's examples, exercises and projects—especially in the file-processing chapter—you'll work with **real-world data** such as Shakespeare's play *Romeo and Juliet*. You'll download and analyze text from Project Gutenberg—a great source of free downloadable texts for analysis. The site contains nearly 63,000 e-books in various formats, including plain-text files—these are out of copyright in the United States. You'll also work with real-world temperature data. In particular, you'll analyze 126 years of New York City average January temperature data and determine whether there is a cooling or warming trend. You'll get this data from National Oceanic and Atmospheric Administration (NOAA) website noaa.gov.

Thinking Like a Developer—GitHub, StackOverflow and More

The best way to prepare [to be a programmer] is to write programs, and to study great programs that other people have written. In my case, I went to the garbage cans at the Computer Science Center and fished out listings of their operating systems. 12

—William Gates

- To help prepare for your career, you'll work with such popular developer websites as **GitHub** and **StackOverflow**, and you'll do Internet research.
- StackOverflow is one of the most popular developer-oriented, question-andanswer sites.
- There is a massive C open-source community. For example, on **GitHub**, there are over 32,000¹³ C code repositories! You can check out other people's C code on GitHub and even build upon it if you like. This is a great way to learn and is a natural extension of our live-code teaching philosophy.¹⁴
- GitHub is an excellent venue for finding free, open-source code to incorporate into your projects—and for you to contribute your code to the open-

^{12.} William Gates, quoted in *Programmers at Work: Interviews With 19 Programmers Who Shaped the Computer Industry* by Susan Lammers. Microsoft Press, 1986, p. 83.

^{13. &}quot;C." Accessed January 4, 2021. https://github.com/topics/c.

^{14.} Students will need to become familiar with the variety of open-source licenses for software on GitHub.

xxxii Preface

source community if you like. Fifty million developers use GitHub.¹⁵ The site currently hosts over 100 million repositories for code written in an enormous number of languages¹⁶—developers contributed to 44+ million repositories in 2019 alone.¹⁷ GitHub is a crucial element of the professional software developer's arsenal with version control tools that help teams of developers manage public open-source projects and private projects.

- In 2018, Microsoft purchased GitHub for \$7.5 billion. If you become a soft-ware developer, you'll almost certainly use GitHub regularly. According to Microsoft's CEO, Satya Nadella, they bought GitHub to "empower every developer to build, innovate and solve the world's most pressing challenges." 18
- We encourage you to study and execute lots of developers' open-source C code on GitHub.

Privacy

The ACM/IEEE's curricula recommendations for Computer Science, Information Technology and Cybersecurity mention privacy over 200 times. Every programming student and professional needs to be acutely aware of privacy issues and concerns. Students research privacy in four exercises in Chapters 1, 3 and 10.

In Chapter 1's exercises, you'll start thinking about these issues by researching everstricter privacy laws such as HIPAA (Health Insurance Portability and Accountability Act) and the California Consumer Privacy Act (CCPA) in the United States and GDPR (General Data Protection Regulation) for the European Union.

Ethics

The ACM's curricula recommendations for Computer Science, Information Technology and Cybersecurity mention ethics more than 100 times. In several Chapter 1 exercises, you'll focus on ethics issues via Internet research. You'll investigate privacy and ethical issues surrounding intelligent assistants, such as IBM Watson, Amazon Alexa, Apple Siri, Google Assistant and Microsoft Cortana. For example, a judge ordered Amazon to turn over Alexa recordings for use in a criminal case.¹⁹

Performance

Programmers prefer C (and C++) for performance-intensive operating systems, real-time systems, embedded systems, game systems and communications systems, so we focus on performance issues. We use timing operations in our multithreading exam-

^{15. &}quot;GitHub." Accessed November 14, 2020. https://github.com/.

^{16. &}quot;GitHub is how people build software." Accessed November 14, 2020. https://github.com/about.

^{17. &}quot;The State of the Octoverse." Accessed November 14, 2020. https://octoverse.github.com.

^{18. &}quot;Microsoft to acquire GitHub for \$7.5 billion." Accessed November 14, 2020. https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/.

^{19. &}quot;Judge orders Amazon to turn over Echo recordings in double murder case." Accessed November 14, 2020. https://techcrunch.com/2018/11/14/amazon-echo-recordings-judge-murder-case/.

ples to measure the performance improvement we get on today's popular multicore systems, as we employ an increasing number of cores.

Static Code Analysis Tools

Static code analysis tools let you quickly check your code for common errors and security problems and provide insights for improving your code. We checked all our C code using the clang-tidy tool (https://clang.llvm.org/extra/clang-tidy/). We also used the compiler flag -Wall in the GNU gcc and Clang compilers to enable all compiler warnings. With a few exceptions for warnings beyond this book's scope, we ensure that our programs compile without warning messages.

How We're Handling CII's Annex K and printf_s/scanf_s

The C11 standard's Annex K introduced more secure versions of printf (for output) and scanf (for input) called printf_s and scanf_s. We discuss these functions and the corresponding security issues in Sections 6.13 and 7.13:

- Annex K is optional, so not every C vendor implements it. In particular, GNU C++ and Clang C++ do not implement Annex K, so using scanf_s and printf_s might compromise your code's portability among compilers.
- Microsoft implemented its own Visual C++ versions of printf_s and scanf_s before the C11 standard. Its compiler immediately began warning on every scanf call that scanf was deprecated—i.e., it should no longer be used—and that you should consider using scanf_s instead. Microsoft now treats what used to be a warning about scanf as an error. By default, a program with scanf will not compile on Visual C++. Chapter 1's Visual C++ test-drive shows how to handle this issue and compile our programs.
- Many organizations have coding standards that require code to compile without warning messages. There are two ways to eliminate Visual C++'s scanf warnings—use scanf_s instead of scanf or disable these warnings.
- There is some discussion of removing Annex K from the C standard. For this reason, we use printf/scanf throughout this book and show Visual C++ users how to disable Microsoft's printf/scanf errors. Windows users who prefer not to do that can use the gcc compiler in the GNU GCC Docker container, which we discuss in this Preface's "Docker" section. See the Before You Begin section that follows this Preface, and see Section 1.10 for details.

New Appendix: Intro to Object-Oriented Programming Appendix

C's programming model is called **procedural programming**. We teach it as **structured procedural programming**. After learning C, you'll likely also learn one or more C-based object-oriented languages—such as Java, C++, C#, Objective-C or Swift—and use them side-by-side with C. Many of these languages support several programming paradigms among procedural programming, object-oriented programming, generic programming and functional-style programming. In Appendix D, we present a friendly overview of object-oriented programming fundamentals.

A Case Studies Tour

We include many case studies as more substantial chapter examples, exercises and projects (EEPs). These are at an **appropriate level for introductory programming courses**. We anticipate that instructors will select subsets of the case studies appropriate for their particular courses.

Chapter 5—Random-Number Simulation: Building a Casino Game

In this case study, you'll use random-number generation and simulation techniques to implement the popular casino dice game called craps.

Chapter 5—Random-Number Simulation Case Study: The Tortoise and the Hare Race

In this case study exercise, you'll use random-number generation and simulation techniques to implement the famous race between the tortoise and the hare.

Chapter 6—Visualizing Binary Search

In this case study, you'll learn the high-speed binary-search algorithm and see a visualization that shows the algorithm's halving effect that achieves high performance.

Chapter 6—Intro to Data Science: Survey Data Analysis

In this case study, you'll learn various basic descriptive statistics (mean, median and mode) that are commonly used to "get to know your data." You'll then build a nice array-manipulation application that calculates these statistics for a batch of survey data.

Chapter 7—Random-Number Simulation—Card Shuffling and Dealing

In this case study, you'll use arrays of strings, random-number generation and simulation techniques to implement a text-based card-shuffling-and-dealing program.

Chapter 7—Embedded Systems Programming: Robotics with the Webots Simulator

Webots (https://cyberbotics.com/) is a wonderful open-source, 3D, robotics simulator that runs on Windows, macOS and Linux. It comes bundled with simulations for dozens of robots that walk, fly, roll, drive and more:

https://cyberbotics.com/doc/guide/robots

You'll use the free tier of the Webot robotics simulator to explore their dozens of simulated robots. You'll execute various full-color 3D robotics simulations written in C and study the provided code. Webots is a self-contained development environment that provides a C code editor and compiler. You'll use these tools to program your own simulations using Webot's robots.

Webots provides lots of fully coded C programs. A great way for you to learn C is to study existing programs, modify them to work a bit differently and observe the results. Many prominent robotics companies use Webots simulators to prototype new products.

Chapter 7—Systems Software Case Study: Building Your Own Computer (Virtual Machine) with Simulation

In the context of several exercises, you'll "peel open" a *hypothetical* computer and look at its internal structure. We introduce simple machine-language programming and write several small machine-language programs for this computer, which we call the Simpletron. As its name implies, it's a simple machine, but as you'll see, a powerful one as well. The Simpletron runs programs written in the only language it directly understands—that is, Simpletron Machine Language, or SML for short. To make this an especially valuable experience, you'll then build a computer (through the technique of software-based simulation) on which you can actually run your machine-language programs! The Simpletron experience will give you a basic introduction to the notion of virtual machines—one of the most important systems-architecture concepts in modern computing.

Chapter 8—Pqyoaf X Nylfomigrob Qwbbfmh Mndogvk: Rboqlrut yua Boklnxhmywex

This case study exercise's title looks like gibberish. This is not a mistake! In this exercise, we introduce **cryptography**, which is critically important in today's connected world. Every day, cryptography is used behind the scenes to **ensure that your Internet-based communications are private and secure**. This case study exercise continues our security emphasis by having readers study the **Vigenère secret-key cipher** algorithm and implement it using array-processing techniques.²⁰ They'll then use it to encrypt and decrypt messages and to decrypt this section's title.

Chapter 8—RSA Public-Key Cryptography

Secret key encryption and decryption have a weakness—an encrypted message can be decrypted by anyone who discovers or steals the secret key. We explore public-key cryptography with the RSA algorithm. This technique performs encryption with a public key known to every sender who might want to send a secret message to a particular receiver. The public key can be used to encrypt messages but not decrypt them. Messages can be decrypted only with a paired private key known only to the receiver, so it's much more secure than the secret key in secret-key cryptography. RSA is among the world's most widely used public-key cryptography technologies. You'll build a working, small-scale, classroom version of the RSA cryptosystem.

Chapter 10—Raylib Game Programming Case Studies

In this series of five case study exercises and 10 additional exercises, you'll use the open-source, cross-platform raylib²¹ game programming library, which supports Windows, macOS, Linux and other platforms. The raylib development team provides many C demos to help you learn key library features and techniques. You'll study two completely coded games and a dynamic animated visualization that we created:

^{20. &}quot;Vigenère Cipher." Accessed November 22, 2020. https://en.wikipedia.org/wiki/Vigenère_cipher.

^{21. &}quot;raylib." Accessed November 14, 2020. https://www.raylib.com.

xxxvi Preface

- The Spot-On game tests your reflexes by requiring you to click moving spots before they disappear. With each new game level the spots move faster, making them harder to click.
- The Cannon game challenges you to repeatedly aim and fire a cannon to destroy nine moving targets before a time limit expires. A moving blocker makes the game more difficult.
- The Law of Large Numbers dynamic animated visualization repeatedly rolls a six-sided die and creates an **animated bar chart**. **Visualizations** give you a powerful way to understand data that goes beyond simply looking at raw data. This case study exercise allows students to see the "law of large numbers" at work. When repeatedly rolling a die, we expect each die face to appear approximately 1/6th (16.667%) of the time. For small numbers of rolls (e.g., 60 or 600), you'll see that the frequencies typically are not evenly distributed. As you simulate larger numbers of die rolls (e.g., 60,000), you'll see the die frequencies become more balanced. When you simulate significant numbers of die rolls (e.g., 60,000,000), the bars will appear to be the same size.

The games and simulation use various raylib capabilities, including shapes, colors, sounds, animation, collision detection and user-input events (such as mouse clicks).

After studying our code, you'll use the raylib graphics, animation and sound features you learn to enhance your implementation of Chapter 5's Tortoise and the Hare Race. You'll incorporate a traditional horse race's sounds, and multiple tortoise and hare images to create a fun, animated multimedia "extravaganza." Then, you'll use a raylib to enhance this chapter's high-performance card-shuffling-and-dealing simulation to display card images. Finally, you can select from 10 additional raylib game-programming and simulation exercises. Get creative—have some fun designing and building your own games, too!

Chapter 11—Case Study: Building a Random-Access Transaction-Processing System

In this case study, you'll use random-access file processing to implement a simple transaction-processing system that simulates the kind of high-speed direct-access capabilities that industrial-strength database-management systems have. This case study gives you both application-programming and some "under-the-hood" systems-programming experience.

Chapter 11—Artificial Intelligence Case Study: Natural Language Processing (NLP)

Natural Language Processing (NLP) helps computers understand, analyze and process text. One of its most common uses is sentiment analysis—determining whether text has positive, neutral or negative sentiment. Another interesting use of NLP is assessing text readability, which is affected by the vocabulary used, word lengths, sentence structure, sentence lengths, topic and more. While writing this book, we used

the paid (NPL) tool Grammarly²² to help tune the writing and ensure the text's readability for a wide audience. Instructors who use the "flipped classroom" format prefer textbooks that students can understand on their own.

Some people believe that the works of William Shakespeare actually might have been written by Christopher Marlowe or Sir Francis Bacon among others. ^{23,24} In the NLP case study exercise, you'll use array-, string- and file-processing techniques to perform **simple similarity detection** on Shakespeare's *Romeo and Juliet* and Marlowe's *Edward the Second* to determine how alike they are. You may be surprised by the results.

Chapter II—Artificial Intelligence Case Study: Machine Learning with the GNU Scientific Library

Statistics can be deceiving. Dramatically different datasets can have identical or nearly identical descriptive statistics. You'll consider a famous example of this phenomenon—Anscombe's Quartet²⁵—which consists of four datasets of x-y coordinate pairs that differ significantly, yet have nearly identical descriptive statistics. You'll then study a completely coded example that uses the machine-learning technique called **simple linear regression** to calculate the equation of a straight line (y = mx + b)that, given a collection of points (x-y coordinate pairs) representing an independent *variable* (x) and a *dependent variable* (y), describes the relationship between these variables with a straight line, known as the regression line. As you'll see, the regression lines for Anscombe's Quartet are visually identical for all four quite different datasets. The program you'll study then passes commands to the open-source gnuplot package to create several attractive visualizations. gnuplot uses its own plotting language different from C, so in our code, we provide extensive comments that explain its commands. Finally, the case study asks you to run a simple linear regression on 126 years of New York City average January temperature data to determine if there is a cooling or warming trend. As part of this case study, you'll also read comma-separated values (CSV) text files containing the datasets.

Chapter I I—Web Services and the Cloud: Getting a Weather Report Using libcurl and the OpenWeatherMap Web Services; Introducing Mashups

More and more computing today is done "in the cloud," using software and data distributed across the Internet worldwide. The apps we use daily are heavily dependent on various cloud-based services. A service that provides access to itself over the Internet is known as a **web service**. In this case study exercise, you'll work through a completely coded application that uses the **open-source** C **library libcurl** to invoke an

^{22.} Grammarly has free and paid versions (https://www.grammarly.com). They provide free plugins you can use in several popular web browsers.

^{23. &}quot;Did Shakespeare Really Write His Own Plays?" Accessed November 13, 2020. https://www.history.com/news/did-shakespeare-really-write-his-own-plays.

^{24. &}quot;Shakespeare authorship question." Accessed November 13, 2020. https://en.wikipedia.org/wiki/Shakespeare_authorship_question.

^{25. &}quot;Anscombe's quartet." Accessed November 13, 2020. https://en.wikipedia.org/wiki/Anscombe%27s_quartet.

xxxviii Preface

OpenWeatherMap (free tier) web service that returns the current weather for a specified city. The web service returns results in JSON (JavaScript Object Notation) format, which we process using the open-source cJSON library.

This exercise opens a world of possibilities. You can explore nearly 24,000 web services listed in the ProgrammableWeb²⁶ web services directory. Many are free or provide free tiers that you can use to create fun and interesting mashups that combine complementary web services.

Chapter 12—Systems Software Case Study: Building Your Own Compiler

In the context of several exercises, you'll build a simple compiler that translates programs written in a simple high-level programming language to our Simpletron Machine Language (SML). You'll write programs in this small new high-level language, compile them on the compiler you build and run them on your Simpletron simulator. And with Chapter 11, File Processing, your compiler can write the generated machine-language code into a file from which your Simpletron computer can then read your SML program, load it into the Simpletron's memory and execute it! This is a nice end-to-end exercise sequence for novice computing students.

Chapter 13—Visualizing the High-Performance Merge Sort

A centerpiece of our sorting treatment is our implementation of the high-performance merge sort algorithm. In that case study, you'll use outputs to visualize the algorithm's partition and merge steps, which will help a user understand how the merge sort works.

Appendix C—Systems Architecture Case Study: Performance with Multithreading and Multicore Systems

Multithreading—which allows you to break a program into separate "threads" that can be executed in parallel—has been around for many decades, but interest in it is higher today due to the availability of multicore processors in computers and devices, including smartphones and tablets. These processors economically implement multiple processors on one integrated circuit chip. They put multiple cores to work executing different parts of your program in parallel, thereby enabling the individual tasks and the program as a whole to complete faster. Four and eight cores are common in many of today's devices, and the number of cores will continue to grow. We wrote and tested the code for this book using an eight-core MacBook Pro. Multithreaded applications enable you to execute separate threads simultaneously on multiple cores, so that you can take the fullest advantage of multicore architecture.

For a convincing demonstration of the power of multithreading on a multicore system, we present a case study with two programs. One performs two compute-intensive calculations in sequence. The other executes the same compute-intensive calculations in parallel threads. We time each calculation and determine the total execution time in each program. The program outputs show the dramatic time improvement when the multithreaded version executes on a multicore system.

^{26. &}quot;ProgrammableWeb." Accessed November 22, 2020. https://programmableweb.com/.

Secure C Programming

The people responsible for the ACM/IEEE curricula guidelines emphasize the importance of security—it's mentioned 395 times in the Computer Science Curricula document and 235 times in the Information Technology Curricula document. In 2017, the ACM/IEEE published its Cybersecurity Curricula, which focuses on security courses and security throughout the other computing curricula. That document mentions security 865 times.

Chapters 2–12 and 14 each end with a Secure C Programming section. These are designed to raise awareness among novice programming students of security issues that could cause breaches. These sections present some key issues and techniques and provide links and references so you can continue learning. Our goal is to encourage you to start thinking about security issues, even if this is your first programming course.

Experience has shown that it's challenging to build industrial-strength systems that stand up to attacks. Today, via the Internet, such attacks can be instantaneous and global in scope. Software vulnerabilities often come from simple programming issues. Building security into software from the start of the development cycle can significantly reduce vulnerabilities.

The CERT Division of Carnegie Mellon's Software Engineering Institute

https://www.sei.cmu.edu/about/divisions/cert/index.cfm

was created to analyze and respond promptly to attacks. They publish and promote secure coding standards to help C programmers and others implement industrial-strength systems that avoid the programming practices that leave systems vulnerable to attacks. Their standards evolve as new security issues arise.

We explain how to upgrade your code (as appropriate for an introductory book) to conform to the latest secure C coding recommendations. If you're building C systems in industry, consider reading the *SEI CERT C Coding Standard* rules at

https://wiki.sei.cmu.edu/confluence/display/c

Also, consider reading *Secure Coding in C and C++, 2/e* by Robert Seacord (Addison-Wesley Professional, 2013). Mr. Seacord, a technical reviewer for an earlier edition of this book, provided specific recommendations on each of our Secure C Programming sections. At the time, he was the Secure Coding Manager at CERT and an adjunct professor at Carnegie Mellon's School of Computer Science. He is now a Technical Director at NCC Group (an IT Security company).

Our Secure C Programming sections discuss many important topics, including:

- Testing for Arithmetic Overflows
- The More Secure Functions in the C Standard's Annex K
- The Importance of Checking the Status Information Returned by Standard-Library Functions
- Range Checking
- Secure Random-Number Generation

xI Preface

- Array Bounds Checking
- Preventing Buffer Overflows
- Input Validation
- Avoiding Undefined Behaviors
- Choosing Functions That Return Status Information vs. Using Similar Functions That Do Not
- Ensuring That Pointers Are Always Null or Contain Valid Addresses
- Using C Functions vs. Using Preprocessor Macros, and More.

Computing and Data Science Curricula

This book is designed for courses that adhere to one or more of the following ACM/ IEEE CS-and-related curriculum documents:

- CC2020: Paradigms for Future Computing Curricula (cc2020.net),²⁷
- Computer Science Curricula 2013,²⁸
- Information Technology Curricula 2017,²⁹
- Cybersecurity Curricula 2017.³⁰

Computing Curricula

- According to "CC2020: A Vision on Computing Curricula,"³¹ the curriculum "needs to be reviewed and updated to include the new and emerging areas of computing such as cybersecurity and data science."³² (See "Data Science Overlaps with Computer Science" below and this Preface's earlier "Secure C Programming" section).
- Data science includes key topics (besides statistics and general-purpose programming) such as machine learning, deep learning, natural language processing, speech synthesis and recognition, and others that are classic artificial intelligence (AI) topics—and hence CS topics as well. We cover machine learning and natural language processing in the case studies.

^{27. &}quot;Computing Curricula 2020." Accessed November 22, 2020. https://cc2020.nsparc.ms-state.edu/wp-content/uploads/2020/11/Computing-Curricula-Report.pdf.

^{28.} ACM/IEEE (Assoc. Comput. Mach./Inst. Electr. Electron. Eng.). 2013. Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science (New York: ACM), http://ai.stanford.edu/users/sahami/CS2013/final-draft/CS2013-final-report.pdf.

^{29.} Information Technology Curricula 2017, http://www.acm.org/binaries/content/assets/education/it2017.pdf.

^{30.} Cybersecurity Curricula 2017, https://cybered.hosting.acm.org/wp-content/uploads/2018/02/newcover_csec2017.pdf.

^{31.} A. Clear, A. Parrish, G. van der Veer and M. Zhang, "CC2020: A Vision on Computing Curricula," https://dl.acm.org/citation.cfm?id=3017690.

^{32.} http://delivery.acm.org/10.1145/3020000/3017690/p647-clear.pdf.

Data Science Overlaps with Computer Science³³

The undergraduate data science curriculum proposal³⁴ includes algorithm development, programming, computational thinking, data structures, database, mathematics, statistical thinking, machine learning, data science and more—a significant overlap with computer science, especially given that the data science courses include some key AI topics. Even though ours is a C programming textbook, we work data science topics into various examples, exercises, projects and case studies.

Key Points from the Data Science Curriculum Proposal

This section calls out some key points from the data science undergraduate curriculum proposal and its detailed course descriptions appendix.³⁵ Each of the following items is covered in *C How to Program*, *9le*:

- Learn programming fundamentals commonly presented in computer science courses, including working with data structures.
- Be able to solve problems by creating algorithms.
- Work with procedural programming.
- Explore concepts via simulations.
- Use development environments (we tested all our code on Microsft Visual C++, Apple Xcode, the GNU command-line gcc compiler on Linux and in the GNU Compiler Collection Docker container).
- Work with real-world data in practical case studies and projects—such as William Shakespeare's Romeo and Juliet and Christopher Marlowe's Edward the Second from Project Gutenberg (https://www.gutenberg.org/), and 126 years of New York City average January temperatures.
- Create data visualizations.
- Communicate **reproducible results**. (**Docker** plays an important role in that—see the next page.)
- Work with existing software and cloud-based tools.
- Work with high-performance tools, such as C's multithreading libraries.
- Focus on data's ethics, security, privacy and reproducibility issues.

^{33.} This section is intended primarily for data science instructors. Given that the emerging 2020 Computing Curricula for computer science and related disciplines is likely to include some key data science topics, this section includes important information for computer science instructors as well.

^{34. &}quot;Curriculum Guidelines for Undergraduate Programs in Data Science," http://www.annual-reviews.org/doi/full/10.1146/annurev-statistics-060116-053930.

^{35. &}quot;Appendix—Detailed Courses for a Proposed Data Science Major," http://www.annualre-views.org/doi/suppl/10.1146/annurev-statistics-060116-053930/suppl_file/st04_de_veaux_supmat.pdf.

Get the Code Examples and Install the Software

For your convenience, we provide the book's examples in C source-code (.c) files for use with integrated development environments (IDEs) and command-line compilers. See the Before You Begin section that follows the Preface for software installation details. See the **Chapter 1 test-drives** for information on running the book's code examples. If you encounter a problem, you can reach us at deitel@deitel.com or via the contact form at https://deitel.com/contact-us.

Docker

We introduce **Docker**—a tool for packaging software into **containers** that bundle everything required to execute that software conveniently, **reproducibly** and **portably** across platforms. Some software packages you'll use require complicated setup and configuration. For many of these, you can download free preexisting **Docker containers** that help you avoid complex installation issues. You can simply execute software locally on your desktop or notebook computers, making Docker a great way to help you get started with new technologies quickly, conveniently and economically. For your convenience, we show how to install and execute a Docker container that's **preconfigured** with the **GNU Compiler Collection** (GCC), which includes the **gcc compiler**. This can run in Docker on **Windows**, **macOS** and **Linux**. It's particularly useful for people using Visual C++, which can compile C code but is not 100% compliant with the latest C standard.

Docker also helps with reproducibility. Custom Docker containers can be configured with every piece of software and every library you use. This would enable others to recreate the environment you used, then reproduce your work, and will help you reproduce your own results. Reproducibility is especially important in the sciences and medicine—for example, when researchers want to prove and extend the work in published articles.

Flipped Classrooms

Many instructors use "flipped classrooms." Students learn the content on their own before coming to class, and class time is used for tasks such as hands-on coding, working in groups and discussions. Our book and supplements also are appropriate for flipped classrooms:

- We use Grammarly to control the book's reading level to help ensure it's appropriate for students learning on their own.
- In parallel with reading the text, students should execute the 147 live-code C examples and do the 350+ integrated Self Check exercises, which are imme-

^{36.} https://en.wikipedia.org/wiki/Flipped_classroom.

^{37.} https://www.edsurge.com/news/2018-05-24-a-case-for-flipping-learning-without-videos.

diately followed by their answers. These encourage active participation by the student. They learn the content in small pieces using a "read-a-little, do-a-little, test-a-little" approach—appropriate for a flipped classroom's active, self-paced, hands-on learning. Students are encouraged to modify the code and see the effects of their changes.

- We provide 445 exercises and projects, which students can work on at home and/or in class. Many of the exercises are at an elementary or intermediate level that students should be able to do independently. And many are appropriate for group projects on which students can collaborate in class.
- Section-by-section detailed chapter summaries with bolded key terms help students quickly review the material.
- In the book's extensive index, the **defining occurrences** of key terms are highlighted with a **bold** page number, making it easy for students to find the introductions to the topics they're studying. This facilitates the outside-the-classroom learning experience of the **flipped classroom**.

A key aspect of flipped classrooms is getting your questions answered when you're working on your own. See the "Getting Your Questions Answered" section later in this Preface for details. And you can always reach us at deitel@deitel.com.

Teaching Approach

C How to Program, 9/e contains a rich collection of examples, exercises, projects and case studies drawn from many fields. Students solve interesting, real-world problems working with real-world data. The book concentrates on the principles of good software engineering and stresses program clarity.

Using Fonts for Emphasis

We place the key terms and the index's page reference for each defining occurrence in **bold text** for easier reference. C code uses a fixed-width font (e.g., x = 5). We place on-screen components in the **bold Helvetica** font (e.g., the File menu).

Syntax Coloring

For readability, we syntax color all the code. In our full-color books and e-books, our syntax-coloring conventions are as follows:

```
comments appear in green
keywords appear in dark blue
constants and literal values appear in light blue
errors appear in red
all other code appears in black
```

Objectives and Outline

Each chapter begins with objectives that tell you what to expect and give you an opportunity, after reading the chapter, to determine whether it has met the intended goals. The chapter outline enables students to approach the material in a top-down fashion.

Examples

The book's 147 live-code examples contain thousands of lines of proven code.

Tables and Illustrations

Abundant tables and line drawings are included.

Programming Wisdom

We integrate into the text discussions programming wisdom and mistakes we've accumulated from our combined nine decades of programming and teaching experience, and from the scores of academics and industry professionals who have reviewed the nine editions of this book over the past 29 years, including:

- Good programming practices and preferred C idioms that help you produce clearer, more understandable and more maintainable programs.
- Common programming errors to reduce the likelihood that you'll make them.
- Error-prevention tips with suggestions for exposing bugs and removing them from your programs. Many of these tips describe techniques for preventing bugs from getting into your programs in the first place.
- **Performance tips** highlighting opportunities to make your programs run faster or minimize the amount of memory they occupy.
- Software engineering observations highlighting architectural and design issues for proper software construction, especially for larger systems.
- Security best practices that will help you strengthen your programs against attacks.

Section-By-Section Chapter Summaries

To help students quickly review the material, each chapter ends with a detailed bulletlist summary with **bolded** key terms and, for most, **bold** page references to their defining occurrences.

Free Software Used in the Book

The **Before You Begin** section following this Preface discusses installing the software you'll need to work with our examples. We tested *C How to Program, 9/e*'s examples using the following popular *free* compilers:

- GNU gcc on Linux—which is already installed on most Linux systems and can be installed on macOS and Windows systems.
- Microsoft's Visual Studio Community Edition on Windows.
- Apple's Clang compiler in Xcode on macOS.

GNU gcc in Docker

We also demonstrate GNU gcc in a Docker container—ideal for instructors who want all their students to use GNU gcc, regardless of their operating system. This









gives Visual C++ users a true C compiler option, since Visual C++ is not 100% compliant with the latest C standard.

Windows Subsystem for Linux

The Windows Subsystem for Linux (WSL) enables Windows users to install Linux and run it inside Windows. We provide a link to Microsoft's step-by-step instructions for setting up WSL and installing a Linux distribution. This provides yet another option for Windows users to access the GNU gcc compiler.

C Documentation

You'll find the following documentation helpful as you work through the book:

- The GNU C Standard Library Reference Manual:
 - https://www.gnu.org/software/libc/manual/pdf/libc.pdf
- C Language Reference at cppreference.com
 - https://en.cppreference.com/w/c
- C Standard Library Headers at cppreference.com https://en.cppreference.com/w/c/header
- Microsoft's C Language Reference:

https://docs.microsoft.com/en-us/cpp/c-language/c-languagereference

Getting Your Questions Answered

Online forums enable you to interact with other C programmers worldwide and get your questions answered. Popular C and general programming online forums include:

- https://stackoverflow.com
- https://www.reddit.com/r/C_Programming/
- https://groups.google.com/forum/#!forum/comp.lang.C
- https://cboard.cprogramming.com/c-programming/
- https://www.dreamincode.net/forums/forum/15-c-and-c/

For a list of other sites, see

```
https://www.geeksforgeeks.org/stuck-in-programming-get-the-solution-from-these-10-best-websites/
```

Also, vendors often provide forums for their tools and libraries. Many libraries are managed and maintained at **github.com**. Some library maintainers provide support through the **Issues tab** on a given library's GitHub page.

Student and Instructor Supplements

The following supplements are available to students and instructors.

Web-Based Materials on deitel.com

To get the most out of your *C How to Program, 9/e* learning experience, you should execute each code example in parallel with reading the corresponding discussion. On the book's web page at

https://deitel.com/c-how-to-program-9-e

we provide the following resources:

Links to the downloadable C source code (.c files) for the book's code examples and exercises that include code in the exercise description. You can also get this from the book's Pearson companion website at

https://pearson.com/deitel

- Links to our **Getting Started videos** showing how to use the compilers and code examples. We also introduce these tools in **Chapter 1**.
- Blog posts—https://deitel.com/blog.
- Book updates—https://deitel.com/c-how-to-program-9-e.
- "Using the Debugger" appendices for the Visual Studio, GNU gdb and Xcode debuggers.

For more information about downloading the examples and setting up your C development environment, see the **Before You Begin** section that follows this Preface.

Instructor Supplements

Pearson Education's IRC (Instructor Resource Center)

http://www.pearsonhighered.com/irc

provides qualified instructors access to the following supplements for this book:

- PowerPoint Slides.
- Instructor Solutions Manual with solutions to most of the exercises. Solutions are not provided for "project" and "research" exercises. Before assigning a particular exercise for homework, instructors should check the IRC to ensure that the solution is available.
- Test Item File with four-part multiple-choice, short-answer questions and answers. You may also request from your Pearson representative (https://pearson.com/replocator) versions of the Test Item File for use with popular automated assessment tools.

Please do not write to us requesting access to the Pearson Instructor's Resource Center (IRC). Access to the instructor supplements and exercise solutions on the IRC is strictly limited by our publisher to college instructors who adopt the book for their classes. Instructors may obtain access through their Pearson representatives. If you're not a registered faculty member, contact your Pearson representative or visit

https://pearson.com/replocator

Instructors can request examination copies of Deitel books from their Pearson representatives.

Communicating with the Authors

For questions, instructor syllabus assistance or to report an error, we're easy to reach at deitel@deitel.com

or via the contact form at

https://deitel.com/contact-us

Interact with us via social media on

- Facebook®—https://facebook.com/DeitelFan
- Twitter®——@deitel or https://twitter.com/deitel
- LinkedIn®—https://linkedin.com/company/deitel-&-associates
- $YouTube^{\mathbb{R}}$ —https://youtube.com/DeitelTV

Deitel Pearson Products on O'Reilly Online Learning

O'Reilly Online Learning subscribers have access to many Deitel Pearson textbooks, professional books, LiveLessons videos and Full Throttle one-day webinars. Sign up for a 10-day free trial at

https://deitel.com/LearnWithDeitel

Textbooks and Professional Books

Each Deitel e-book on O'Reilly Online Learning is presented in full color and extensively indexed.

Asynchronous LiveLessons Video Products

Learn hands-on with Paul Deitel as he presents compelling, leading-edge computing technologies in Python, Python Data Science/AI and Java. C++20 and C are coming in 2021.

Live Full Throttle Webinars

Paul Deitel offers Full Throttle webinars at O'Reilly Online Learning. These are one-full-day, fast-paced, code-intensive introductions to Python, Python Data Science/AI and Java, with C++20 and C coming in 2021. Paul's Full Throttle webinars are for experienced developers and software project managers preparing for projects using other languages. After taking a Full Throttle course, participants often take the corresponding LiveLessons video course which has many more hours of classroom-paced learning.

Acknowledgments

We'd like to thank Barbara Deitel for long hours devoted to Internet research on this project. We're fortunate to have worked with the dedicated team of publishing professionals at Pearson. We appreciate the guidance, wisdom and energy of Tracy Johnson (Pearson Education, Global Content Manager, Computer Science)—on all our

xlviii Preface

academic publications, both print and digital. She challenges us at every step of the process to "get it right" and make the best books. Carole Snyder managed the book's production and interacted with Pearson's permissions team, promptly clearing our graphics and citations to keep the book on schedule. Erin Sullivan recruited and managed the book's review team. We selected the cover art, and Chuti Prasertsith designed the cover, adding his special touch of graphics magic.

We wish to acknowledge the efforts of our academic and professional reviewers. Adhering to a tight schedule, the reviewers scrutinized the manuscript, providing countless suggestions for improving the presentation's accuracy, completeness and timeliness. They helped us make a better book.

Reviewers

C How to Program, 9/e Reviewers

Dr. Danny Kalev (Ben-Gurion University of the Negev, A Certified System Analyst, C Expert and Former Member of the C++ Standards Committee) José Antonio González Seco (Parliament of Andalusia)

C How to Program, 8/e Reviewers

Dr. Brandon Invergo (GNU/European Bioinformatics Institute)

Jim Hogg (Program Manager, C/C++ Compiler Team, Microsoft Corporation)

José Antonio González Seco (Parliament of Andalusia)

Alan Bunning (Purdue University)

Paul Clingan (Ohio State University)

Michael Geiger (University of Massachusetts, Lowell)

Dr. Danny Kalev (Ben-Gurion University of the

Negev, A Certified System Analyst, C Expert and Former Member of the C++ Standards Committee)

Jeonghwa Lee (Shippensburg University)

Susan Mengel (Texas Tech University)

Judith O'Rourke (SUNY at Albany)

Chen-Chi Shin (Radford University)

Other Recent Editions Reviewers (and their affiliations at the time)

William Albrecht (University of South Florida) Ian Barland (Radford University) Ed James Beckham (Altera)

John Benito (Blue Pilot Consulting, Inc. and Convener of ISO WG14—the Working Group responsible for the C Programming Language Standard)

Dr. John F. Doyle (Indiana University Southeast) Alireza Fazelpour (Palm Beach Community College)

Mahesh Hariharan (Microsoft)

Hemanth H.M. (Software Engineer at SonicWALL)

Kevin Mark Jones (Hewlett Packard)

Lawrence Jones, (UGS Corp.)

Don Kostuch (Independent Consultant)

Vytautus Leonavicius (Microsoft)

Xiaolong Li (Indiana State University)

William Mike Miller (Edison Design Group, Inc.)

Tom Rethard (The University of Texas at Arlington)

Robert Seacord (Secure Coding Manager at SEI/

CERT, author of The CERT C Secure Coding Standard and technical expert for the interna-

tional standardization working group for the programming language C)

Benjamin Seyfarth (University of Southern Mississippi)

Gary Sibbitts (St. Louis Community College at Meramec)

William Smith (Tulsa Community College)

Douglas Walls (Senior Staff Engineer, C compiler, Sun Microsystems—now Oracle).

A special thanks to Prof. Alison Clear, an Associate Professor in the School of Computing at Eastern Institute of Technology (EIT) in New Zealand and co-chair of the Computing Curricula 2020 (CC2020) Task Force, which recently released new computing curricula recommendations:

Computing Curricula 2020: Paradigms for Future Computing Curricula https://cc2020.nsparc.msstate.edu/wp-content/uploads/2020/11/Computing-Curricula-Report.pdf

Prof. Clear graciously answered our questions.

And finally, a special note of thanks to the enormous numbers of technically oriented people worldwide who contribute to the open-source movement and write about their work online, to their organizations that encourage the proliferation of such open software and information, and to Google, whose search engine answers our constant stream of questions, each in a fraction of a second, at any time day or night—and at no charge.

Well, there you have it! As you read the book, we'd appreciate your comments, criticisms, corrections and suggestions for improvement. Please send all correspondence, including questions, to

deitel@deitel.com

We'll respond promptly. Welcome to the exciting world of C programming for the 2020s. We hope you have an informative, entertaining and challenging learning experience with *C How to Program*, *9/e* and enjoy this look at leading-edge software development with C. We wish you great success!

Paul Deitel Harvey Deitel

About the Authors

Paul J. Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is an MIT graduate with 41 years of experience in computing. Paul is one of the world's most experienced programming-languages trainers, having taught professional courses to software developers since 1992. He has delivered hundreds of programming courses to academic, industry, government and military clients internationally, including UCLA, Cisco, IBM, Siemens, Sun Microsystems (now Oracle), Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, Nortel Networks, Puma, iRobot and many more. He and his co-author, Dr. Harvey M. Deitel, are among the world's best-selling programming-language textbook, professional book, video and interactive multimedia e-learning authors, and virtual- and live-training presenters.

Dr. Harvey M. Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has 59 years of experience in computing. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University—he studied computing in each of these programs before they spun off Computer Science programs. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., in 1991 with his son, Paul. The Deitels' publications have earned international recognition,

Preface

with more than 100 translations published in Japanese, German, Russian, Spanish, French, Polish, Italian, Simplified Chinese, Traditional Chinese, Korean, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to academic, corporate, government and military clients.

About Deitel® & Associates, Inc.

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate-training organization, specializing in computer programming languages, object technology, mobile app development and Internet and web software technology. The company's training clients include some of the world's largest companies, government agencies, branches of the military, and academic institutions. The company offers instructor-led training courses delivered virtually and live at client sites worldwide, and for Pearson Education on O'Reilly Online Learning.

Through its 45-year publishing partnership with Pearson/Prentice Hall, Deitel & Associates, Inc., publishes leading-edge programming textbooks and professional books in print and e-book formats, LiveLessons video courses, O'Reilly Online Learning live webinars and RevelTM interactive multimedia college courses.

To contact Deitel & Associates, Inc. and the authors, or to request a proposal for virtual or on-site, instructor-led training worldwide, write to

```
deitel@deitel.com
```

To learn more about Deitel on-site corporate training, visit

```
https://deitel.com/training
```

Individuals wishing to purchase Deitel books can do so at

```
https://amazon.com
```

Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For corporate and government sales, send an email to

```
corpsales@pearsoned.com
```

For textbook orders visit

```
https://pearson.com
```

Deitel e-books are available in various formats from

```
https://www.amazon.com/ https://www.vitalsource.com/ https://www.bn.com/ https://www.redshelf.com/ https://www.informit.com/ https://www.chegg.com/
```

To register for a free 10-day trial to O'Reilly Online Learning, visit

```
https://deitel.com/LearnWithDeitel
```

which will forward you to our O'Reilly Online Learning landing page. On that page, click the Begin a free trial link.



Before You Begin

Before using this book, please read this section to understand our conventions and ensure that your computer can compile and run our example programs.

Font and Naming Conventions

We use fonts to distinguish application elements and C++ code elements from regular text. For on-screen application elements, we use a sans-serif bold font, as in the File menu. For C code elements, we use a sans-serif font, as in sqrt(9).

Obtaining the Code Examples

We maintain the code examples for *C How to Program*, *9/e* in a GitHub repository. The book's web page

https://deitel.com/c-how-to-program-9-e

includes a link to the repository and a link to a ZIP file containing the code. If you download the ZIP file, be sure to extract its contents once the download completes. In our instructions, we assume the examples reside in your user account's Documents folder in a subfolder named examples.

If you're not familiar with Git and GitHub but are interested in learning about these essential developer tools, check out their guides at

https://guides.github.com/activities/hello-world/

Compilers We Use in C How to Program, 9/e

We tested *C How to Program*, *9/e*'s examples using the following free compilers:

- For Microsoft Windows, we used Microsoft Visual Studio Community edition¹, which includes the Visual C++ compiler and other Microsoft development tools. Visual C++ can compile C code.
- For macOS, we used Apple Xcode, which includes the Clang C compiler. Command-line Clang also can be installed on Linux and Windows Systems.
- For Linux, we used the GNU gcc compiler—part of the GNU Compiler Collection (GCC). GNU gcc is already installed on most Linux systems and can be installed on macOS and Windows systems.

^{1.} At the time of this writing, the current version was Visual Studio 2019 Community edition.

This Before You Begin section describes installing the compilers. Section 1.10's test-drives demonstrate how to compile and run C programs using these compilers.

Before You Begin Videos

To help you get started with each of our preferred compilers, we provide Before You Begin videos at:

```
https://deitel.com/c-how-to-program-9-e
```

We also provide a Before You Begin video demonstrating how to install the GNU GCC Docker container. This enables you to use the gcc compiler on any Docker-enabled computer.² See the section, "Docker and the GNU Compiler Collection (GCC) Docker Container" later in this Before You Begin section.

Installing Visual Studio Community Edition on Windows

If you use Windows, first ensure that your system meets the requirements for Microsoft Visual Studio Community edition at:

https://docs.microsoft.com/en-us/visualstudio/releases/2019/system-requirements

Next, go to:

```
https://visualstudio.microsoft.com/downloads/
```

then perform the following installation steps:

- 1. Click Free download under Community.
- 2. Depending on your web browser, you may see a pop-up at the bottom of your screen in which you can click **Run** to start the installation process. If not, double-click the installer file in your **Downloads** folder.
- 3. In the User Account Control dialog, click Yes to allow the installer to make changes to your system.
- **4.** In the **Visual Studio Installer** dialog, click **Continue** to allow the installer to download the components it needs for you to configure your installation.
- 5. For this book's examples, select the option **Desktop Development with C++**, which includes the Visual C++ compiler and the C and C++ standard libraries.
- **6.** Click **Install**. Depending on your Internet connection speed, the installation process can take a significant amount of time.

Installing Xcode on macOS

On macOS, perform the following steps to install Xcode:

1. Click the Apple menu and select App Store..., or click the App Store icon in the dock at the bottom of your Mac screen.

^{2. &}quot;Docker Frequently Asked Questions (FAQ)." Accessed January 3, 2021. https://docs.docker.com/engine/faq/.

- 2. In the App Store's Search field, type Xcode.
- **3.** Click the **Get** button to install Xcode.

Installing GNU gcc on Linux

Most Linux users already have a recent version of GNU gcc installed. To check, open a shell or Terminal window on your Linux system, then enter the command

```
qcc --version
```

If it does not recognize the command, you must install GNU gcc. We use the Ubuntu Linux distribution. On that distribution, you must be logged in as an administrator or have the administrator password to execute the following commands:

- 1. sudo apt update
- sudo apt install build-essential gdb

Linux distributions often use different software installation and upgrade techniques. If you are not using Ubuntu Linux, search online for "Install GCC on MyLinuxDistribution" and replace MyLinuxDistribution with your Linux version. You can download the GNU Compiler Collection for various platforms at:

https://gcc.gnu.org/install/binaries.html

Installing GNU GCC in Ubuntu Linux Running on the Windows Subsystem for Linux

Another way to install GNU gcc on Windows is via the Windows Subsystem for Linux (WSL), which enables you to run Linux on Windows. Ubuntu Linux provides an easy-to-use installer in the Windows Store, but first you must install WSL:

- 1. In the search box on your taskbar, type "Turn Windows features on or off," then click Open in the search results.
- 2. In the Windows Features dialog, locate Windows Subsystem for Linux and ensure that it is checked. If it is, WSL is already installed. Otherwise, check it and click OK. Windows will install WSL and ask you to reboot your system.
- 3. Once the system reboots and you log in, open the Microsoft Store app and search for Ubuntu, select the app named Ubuntu and click Install. This installs the latest version of Ubuntu Linux.
- **4.** Once installed, click the **Launch** button to display the Ubuntu Linux command-line window, which will continue the installation process. You'll be asked to create a username and password for your Ubuntu installation—these do not need to match your Windows username and password.
- 5. When the Ubuntu installation completes, execute the following two commands to install the GCC and the GNU debugger—you may be asked enter your Ubuntu password for the account you created in Step 6:

```
sudo apt-get update
sudo apt-get install build-essential gdb
```

liv Before You Begin

6. Confirm that gcc is installed by executing the following command:

```
gcc --version
```

To access our code files, use the cd command change the folder within Ubuntu to:

```
cd /mnt/c/Users/YourUserName/Documents/examples
```

Use your own user name and update the path to where you placed our examples on your system.

GNU Compiler Collection (GCC) Docker Container

Docker is a tool for packaging software into containers (also called images) that bundle everything required to execute software across platforms. Docker is particularly useful for software packages with complicated setups and configurations. You typically can download preexisting Docker containers (often at https://hub.docker.com) for free and execute them locally on your desktop or notebook computer. This makes Docker a great way to get started with new technologies quickly and conveniently.

Docker makes it easy to use the GNU Compiler Collection on most versions of Windows 10, and on macOS and Linux. The GNU Docker containers are located at

```
https://hub.docker.com/_/gcc
```

Installing Docker

To use the GCC Docker container, first install Docker. Windows (64-bit)³ and macOS users should download and run the **Docker Desktop** installer from:

```
https://www.docker.com/get-started
```

then follow the on-screen instructions. Linux users should install Docker Engine from:

```
https://docs.docker.com/engine/install/
```

Also, sign up for a **Docker Hub** account on this webpage so you can install pre-configured containers from https://hub.docker.com.

Downloading the Docker Container

Once Docker is installed and running, open a Command Prompt (Windows), Terminal (macOS/Linux) or shell (Linux), then execute the command:

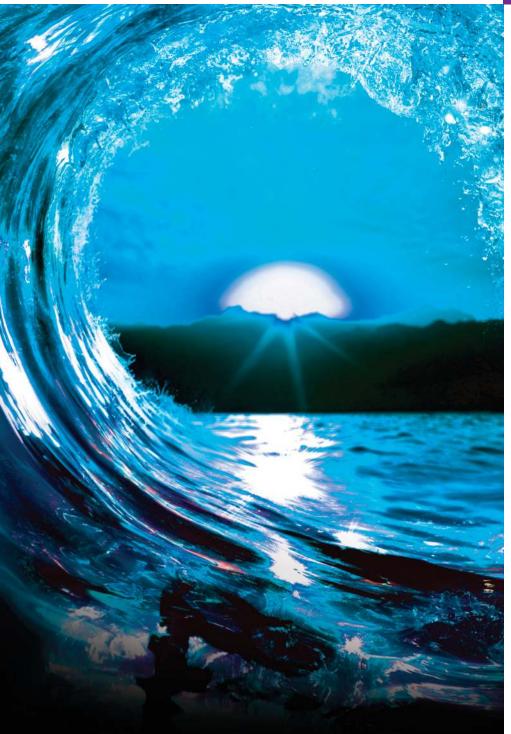
```
docker pull gcc:latest
```

Docker downloads the GNU Compiler Collection (GCC) container's current version.⁴ In one of Section 1.10's test-drives, we'll demonstrate how to execute the container and use it to compile and run C programs.

^{3.} If you have Windows Home (64-bit), follow the instructions at https://docs.docker.com/docker-for-windows/install-windows-home/.

^{4.} At the time of this writing, the current version of the GNU Compiler Collection is 10.2.

Introduction to Computers and C



Objectives

In this chapter, you'll:

- Learn about exciting recent developments in computing.
- Learn computer hardware, software and Internet basics.
- Understand the data hierarchy from bits to databases.
- Understand the different types of programming languages.
- Understand the strengths of C and other leading programming languages.
- Be introduced to the C standard library of reusable functions that help you avoid "reinventing the wheel."
- Test-drive a C program that you compile with one or more of the popular C compilers we used to develop the book's hundreds of C code examples, exercises and projects (EEPs).
- Be introduced to big data and data science.
- Be introduced to artificial intelligence—a key intersection of computer science and data science.

2

- I.I Introduction
- **1.2** Hardware and Software
 - 1.2.1 Moore's Law
 - 1.2.2 Computer Organization
- **1.3** Data Hierarchy
- 1.4 Machine Languages, Assembly Languages and High-Level Languages
- **1.5** Operating Systems
- **1.6** The C Programming Language
- 1.7 The C Standard Library and Open-Source Libraries
- **1.8** Other Popular Programming Languages
- 1.9 Typical C Program-Development Environment
 - 1.9.1 Phase 1: Creating a Program
 - 1.9.2 Phases 2 and 3: Preprocessing and Compiling a C Program
 - 1.9.3 Phase 4: Linking
 - 1.9.4 Phase 5: Loading
 - 1.9.5 Phase 6: Execution
 - 1.9.6 Problems That May Occur at Execution Time
 - 1.9.7 Standard Input, Standard Output and Standard Error Streams
- 1.10 Test-Driving a C Application in Windows, Linux and macOS

- 1.10.1 Compiling and Running a CApplication with Visual Studio 2019Community Edition on Windows 10
- 1.10.2 Compiling and Running a C
 Application with Xcode on macOS
- 1.10.3 Compiling and Running a C Application with GNU gcc on Linux
- 1.10.4 Compiling and Running a C Application in a GCC Docker Container Running Natively over Windows 10, macOS or Linux
- I.II Internet, World Wide Web, the Cloud and IoT
 - I.II.I The Internet: A Network of Networks
 - 1.11.2 The World Wide Web: Making the Internet User-Friendly
 - 1.11.3 The Cloud
 - 1.11.4 The Internet of Things
- **1.12** Software Technologies
- 1.13 How Big Is Big Data?
 - 1.13.1 Big-Data Analytics
 - 1.13.2 Data Science and Big Data Are Making a Difference: Use Cases
- **1.14** Case Study—A Big-Data Mobile Application
- 1.15 Al—at the Intersection of Computer Science and Data Science

Self-Review Exercises | Answers to Self-Review Exercises | Exercises

I.I Introduction

Welcome to C—one of the world's most senior computer programming languages and, according to the Tiobe Index, the world's most popular. You're probably familiar with many of the powerful tasks computers perform. In this textbook, you'll get intensive, hands-on experience writing C instructions that command computers to perform those and other tasks. Software (that is, the C instructions you write, which are also called code) controls hardware (that is, computers and related devices).

^{1. &}quot;TIOBE Index." Accessed November 4, 2020. https://www.tiobe.com/tiobe-index/.

C is widely used in industry for a wide range of tasks.² Today's popular desktop operating systems—Windows³, macOS⁴ and Linux⁵—are partially written in C. Many popular applications are partially written in C, including popular web browsers (e.g., Google Chrome⁶ and Mozilla Firefox⁷), database management systems (e.g., Microsoft SQL Server⁸, Oracle⁹ and MySQL¹⁰) and more.

In this chapter, we introduce terminology and concepts that lay the groundwork for the C programming you'll learn, beginning in Chapter 2. We'll introduce hardware and software concepts. We'll also overview the data hierarchy—from individual bits (ones and zeros) to databases, which store the massive amounts of data that organizations need to implement contemporary applications such as Google Search, Netflix, Twitter, Waze, Uber, Airbnb and a myriad of others.

We'll discuss the types of programming languages. We'll introduce the C standard library and various C-based "open-source" libraries that help you avoid "reinventing the wheel." You'll use these libraries to perform powerful tasks with modest numbers of instructions. We'll introduce additional software technologies that you're likely to use as you develop software in your career.

Many development environments are available in which you can compile, build and run C applications. You'll work through one or more of the four test-drives showing how to compile and execute C code using:

- Microsoft Visual Studio 2019 Community edition for Windows.
- Clang in Xcode on macOS.
- GNU gcc in a shell on Linux.
- GNU gcc in a shell running inside the GNU Compiler Collection (GCC) Docker container.

You can read only the test-drive(s) required for your course or projects in industry.

In the past, most computer applications ran on "standalone" computers (that is, not networked together). Today's applications can communicate among the world's computers via the Internet. We'll introduce the Internet, the World Wide Web, the Cloud and the Internet of Things (IoT), each of which could play a significant part in the applications you'll build in the 2020s (and probably long afterward).

^{2. &}quot;After All These Years, the World is Still Powered by C Programming." Accessed Nov. 4, 2020. https://www.toptal.com/c/after-all-these-years-the-world-is-still-powered-by-c-programming.

^{3. &}quot;What Programming Language is Windows written in?" Accessed Nov. 4, 2020. https://social.microsoft.com/Forums/en-US/65a1fe05-9c1d-48bf-bd40-148e6b3da9f1/what-programming-language-is-windows-written-in.

^{4. &}quot;macOS." Accessed Nov. 4, 2020. https://en.wikipedia.org/wiki/MacOS.

^{5. &}quot;Linux kernel." Accessed Nov. 4, 2020. https://en.wikipedia.org/wiki/Linux_kernel.

^{6. &}quot;Google Chrome." Accessed Nov. 4, 2020. https://en.wikipedia.org/wiki/Google_Chrome.

^{7. &}quot;Firefox." Accessed Nov. 4, 2020. https://en.wikipedia.org/wiki/Firefox.

^{8. &}quot;Microsoft SQL Server." Accessed Nov. 4, 2020. https://en.wikipedia.org/wiki/Microsoft_SQL_Server.

^{9. &}quot;Oracle Database." Accessed Nov. 4, 2020. https://en.wikipedia.org/wiki/Oracle_Database.

^{10. &}quot;MySQL." Accessed Nov. 4, 2020. https://en.wikipedia.org/wiki/MySQL.

1.2 Hardware and Software

Computers can perform calculations and make logical decisions phenomenally faster than human beings can. Today's personal computers and smartphones can perform billions of calculations in one second—more than a human can perform in a lifetime. Supercomputers already perform thousands of trillions (quadrillions) of instructions per second! As of December 2020, Fujitsu's Fugaku¹¹ is the world's fastest supercomputer—it can perform 442 quadrillion calculations per second (442 petaflops)!¹² To put that in perspective, this supercomputer can perform in one second almost 58 million calculations for every person on the planet!¹³ And supercomputing upper limits are growing quickly.

Computers process data under the control of sequences of instructions called computer programs (or simply programs). These programs guide the computer through ordered actions specified by people called computer programmers.

A computer consists of various physical devices referred to as hardware—such as the keyboard, screen, mouse, solid-state disks, hard disks, memory, DVD drives and processing units. Computing costs are dropping dramatically due to rapid developments in hardware and software technologies. Computers that might have filled large rooms and cost millions of dollars decades ago are now inscribed on silicon computer chips smaller than a fingernail, costing perhaps a few dollars each. Ironically, silicon is one of the most abundant materials on Earth—it's an ingredient in common sand. Silicon-chip technology has made computing so economical that computers and computerized devices have become commodities.

1.2.1 Moore's Law

Every year, you probably expect to pay at least a little more for most products and services. The opposite has been the case in the computer and communications fields, especially with regard to the hardware supporting these technologies. Over the years, hardware costs have fallen rapidly.

For decades, every couple of years, computer processing power approximately doubled inexpensively. This remarkable trend often is called Moore's Law, named for Gordon Moore, co-founder of Intel and the person who identified this trend in the 1960s. Intel is a leading manufacturer of the processors in today's computers and embedded systems, such as smart home appliances, home security systems, robots, intelligent traffic intersections and more.

^{11. &}quot;Top 500." Accessed December 24, 2020. https://en.wikipedia.org/wiki/TOP500#TOP_500.

^{12. &}quot;Flops." Accessed November 1, 2020. https://en.wikipedia.org/wiki/FLOPS.

^{13.} For perspective on how far computing performance has come, consider this: In his early computing days in the 1960s, Harvey Deitel used the Digital Equipment Corporation PDP-1 (https://en.wikipedia.org/wiki/PDP-1), which was capable of performing only 93,458 operations per second, and the IBM 1401 (http://www.ibm-1401.info/1401GuidePosterV9.html), which performed only 86,957 operations per second.

Key executives at computer-processor companies NVIDIA and Arm have indicated that Moore's Law no longer applies. 14,15 Computer processing power continues to increase but relies on new processor designs, such as multicore processors (Section 1.2.2).

Moore's Law and related observations apply especially to

- the amount of memory that computers have for programs,
- the amount of secondary storage (such as hard disks and solid-state drive storage) they have to hold programs and data, and
- their processor speeds—that is, the speeds at which computers execute programs to do their work.

Similar growth has occurred in the communications field. Costs have plummeted as enormous demand for communications bandwidth (that is, information-carrying capacity) has attracted intense competition. We know of no other fields in which technology improves so quickly, and costs fall so rapidly. Such phenomenal improvement is truly fostering the Information Revolution.

1.2.2 Computer Organization

Regardless of physical differences, computers can be envisioned as divided into various logical units or sections.

Input Unit

This "receiving" section obtains information (data and computer programs) from input devices and places it at the other units' disposal for processing. Computers receive most user input through keyboards, touch screens, mice and touchpads. Other forms of input include:

- receiving voice commands,
- scanning images and barcodes,
- reading data from secondary storage devices (such as solid-state drives, hard drives, Blu-ray DiscTM drives and USB flash drives—also called "thumb drives" or "memory sticks"),
- receiving video from a webcam,
- receiving information from the Internet (such as when you stream videos from YouTube® or download e-books from Amazon),
- receiving position data from a GPS device,
- receiving motion and orientation information from an accelerometer (a device that responds to up/down, left/right and forward/backward accelera-

^{14. &}quot;Moore's Law turns 55: Is it still relevant?" Accessed November 2, 2020. https://www.techre-public.com/article/moores-law-turns-55-is-it-still-relevant.

^{15. &}quot;Moore's Law is dead: Three predictions about the computers of tomorrow." Accessed November 2, 2020. https://www.techrepublic.com/article/moores-law-is-dead-three-predictions-about-the-computers-of-tomorrow/.

6 Chapter I Introduction to Computers and C

- tion) in a smartphone or wireless game controllers, such as those for Microsoft® Xbox®, Nintendo Switch™ and Sony® PlayStation®, and
- receiving voice input from intelligent assistants like Apple Siri[®], Amazon Alexa[®] and Google Home[®].

Output Unit

This "shipping" section takes information the computer has processed and places it on various output devices to make it available outside the computer. Most information that's output from computers today is

- displayed on screens,
- printed on paper ("going green" discourages this),
- played as audio or video on smartphones, tablets, PCs and giant screens in sports stadiums,
- transmitted over the Internet, or
- used to control other devices, such as self-driving cars (and autonomous vehicles in general), robots and "intelligent" appliances.

Information is also commonly output to secondary storage devices, such as solid-state drives (SSDs), hard drives, USB flash drives and DVD drives. Popular recent forms of output are smartphone and game-controller vibration, virtual reality devices like Oculus Rift[®], Oculus Quest[®], Sony[®] PlayStation[®] VR and Samsung Gear VR[®], and mixed reality devices like Magic Leap[®] One and Microsoft HoloLensTM.

Memory Unit

This rapid-access, relatively low-capacity "warehouse" section retains information entered through the input unit, making it immediately available for processing when needed. The memory unit also retains processed information until it can be placed on output devices by the output unit. Information in the memory unit is volatile—it's typically lost when the computer's power is turned off. The memory unit is often called either memory, primary memory or RAM (Random Access Memory). Main memories on desktop and notebook computers contain as much as 128 GB of RAM, though 8 to 16 GB is most common. GB stands for gigabytes; a gigabyte is approximately one billion bytes. A byte is eight bits. A bit (short for "binary digit") is either a 0 or a 1.

Arithmetic and Logic Unit (ALU)

This "manufacturing" section performs calculations (e.g., addition, subtraction, multiplication and division) and makes decisions (e.g., comparing two items from the memory unit to determine whether they're equal). In today's systems, the ALU is part of the next logical unit, the CPU.

Central Processing Unit (CPU)

This "administrative" section coordinates and supervises the operation of the other sections. The CPU tells

- the input unit when to read information into the memory unit,
- the ALU when to use information from the memory unit in calculations, and
- the output unit when to send information from the memory unit to specific output devices.

Most computers today have multicore processors that economically implement multiple processors on a single integrated circuit chip. Such processors can perform many operations simultaneously. A dual-core processor has two CPUs, a quad-core processor has four and an octa-core processor has eight. Intel has some processors with up to 72 cores.

Secondary Storage Unit

This is the long-term, high-capacity "warehousing" section. Programs and data not actively being used by the other units are placed on secondary storage devices until they're again needed, possibly hours, days, months or even years later. Information on secondary storage devices is persistent—it's preserved even when the computer's power is turned off. Secondary storage information takes much longer to access than information in primary memory, but its cost per byte is much less. Examples of secondary storage devices include solid-state drives (SSDs), USB flash drives, hard drives and read/write Blu-ray drives. Many current drives hold terabytes (TB) of data. A terabyte is approximately one trillion bytes. Typical desktop and notebook-computer hard drives hold up to 4 TB, and some recent desktop-computer hard drives hold up to 20 TB. The largest commercial SSD holds up to 100 TB (and costs \$40,000). The largest commercial SSD holds up to 100 TB (and costs \$40,000).

Self Check

- **I** (*Fill-In*) For many decades, every year or two, computers' capacities have approximately doubled inexpensively. This remarkable trend often is called _____. **Answer:** Moore's Law.
- **2** (*True/False*) Information in the memory unit is persistent—it's preserved even when the computer's power is turned off

Answer: False. Information in the memory unit is volatile—it's typically lost when the computer's power is turned off.

3 *(Fill-In)* Most computers today have ______ processors that implement multiple processors on a single integrated-circuit chip. Such processors can perform many operations simultaneously.

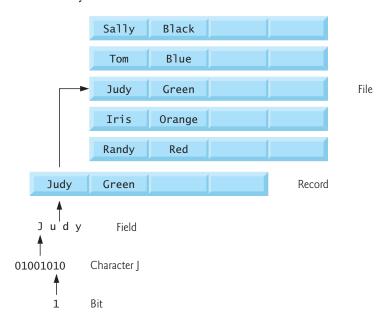
Answer: multicore.

^{16. &}quot;History of hard disk drives." Accessed November 1, 2020. https://en.wikipedia.org/wiki/History_of_hard_disk_drives.

^{17. &}quot;At 100TB, the world's biggest SSD gets an (eye-watering) price tag." Accessed November 1, 2020. https://www.techradar.com/news/at-100tb-the-worlds-biggest-ssd-gets-an-eye-watering-price-tag.

1.3 Data Hierarchy

Data items processed by computers form a data hierarchy that becomes larger and more complex in structure as we progress from the simplest data items (called "bits") to richer ones, such as characters and fields. The following diagram illustrates a portion of the data hierarchy:



Bits

A bit is short for "binary digit"—a digit that can assume one of two values—and is a computer's smallest data item. It can have the value 0 or 1. Remarkably, computers' impressive functions involve only the simplest manipulations of 0s and 1s—examining a bit's value, setting a bit's value and reversing a bit's value (from 1 to 0 or from 0 to 1). Bits form the basis of the binary number system, which we discuss in our "Number Systems" appendix.

Characters

Work with data in the low-level form of bits is tedious. Instead, people prefer to work with decimal digits (0–9), letters (A–Z and a–z) and special symbols such as

Digits, letters and special symbols are known as characters. The computer's character set contains the characters used to write programs and represent data items. Computers process only 1s and 0s, so a computer's character set represents each character as a pattern of 1s and 0s. C uses the ASCII (American Standard Code for Information Interchange) character set by default. C also supports Unicode® characters composed of one, two, three or four bytes (8, 16, 24 or 32 bits, respectively). ¹⁸

^{18. &}quot;Programming with Unicode." Accessed November 1, 2020. https://unicodebook.readthe-docs.io/programming_languages.html.

Unicode contains characters for many of the world's languages. ASCII is a (tiny) subset of Unicode representing letters (a–z and A–Z), digits and some common special characters. You can view the ASCII subset of Unicode at

https://www.unicode.org/charts/PDF/U0000.pdf

For the lengthy Unicode charts for all languages, symbols, emojis and more, visit http://www.unicode.org/charts/

Fields

Just as characters are composed of bits, fields are composed of characters or bytes. A field is a group of characters or bytes that conveys meaning. For example, a field consisting of uppercase and lowercase letters could represent a person's name, and a field consisting of decimal digits could represent a person's age in years.

Records

Several related fields can be used to compose a record. In a payroll system, for example, the record for an employee might consist of the following fields (possible types for these fields are shown in parentheses):

- Employee identification number (a whole number).
- Name (a group of characters).
- Address (a group of characters).
- Hourly pay rate (a number with a decimal point).
- Year-to-date earnings (a number with a decimal point).
- Amount of taxes withheld (a number with a decimal point).

Thus, a record is a group of related fields. All the fields listed above belong to the same employee. A company might have many employees and a payroll record for each.

Files

A file is a group of related records. More generally, a file contains arbitrary data in arbitrary formats. Some operating systems view a file simply as a *sequence of bytes*—any organization of the bytes in a file, such as organizing the data into records, is a view created by the application programmer. You'll see how to do that in Chapter 11, File Processing. It's not unusual for an organization to have many files, some containing billions, or even trillions, of characters of information. As we'll see below, with big data, far larger file sizes are becoming increasingly common.

Databases

A database is a collection of data organized for easy access and manipulation. The most popular model is the relational database, in which data is stored in simple tables. A table includes records and fields. For example, a table of students might include first name, last name, major, year, student ID number and grade-point-average fields. The data for each student is a record, and the individual pieces of informa-

tion in each record are the fields. You can search, sort and otherwise manipulate the data based on its relationship to multiple tables or databases. For example, a university might use data from the student database combined with data from databases of courses, on-campus housing, meal plans, etc.

Big DataThe table below shows some common byte measures:

Unit	Bytes	Which is approximately
1 kilobyte (KB)	1024 bytes	10 ³ bytes (1024 bytes exactly)
1 megabyte (MB)	1024 kilobytes	10 ⁶ (1,000,000) bytes
1 gigabyte (GB)	1024 megabytes	10 ⁹ (1,000,000,000) bytes
1 terabyte (TB)	1024 gigabytes	10 ¹² (1,000,000,000,000) bytes
1 petabyte (PB)	1024 terabytes	10 ¹⁵ (1,000,000,000,000,000) bytes
1 exabyte (EB)	1024 petabytes	$10^{18} (1,000,000,000,000,000,000)$ bytes
1 zettabyte (ZB)	1024 exabytes	10 ²¹ (1,000,000,000,000,000,000,000) bytes

The amount of data being produced worldwide is enormous, and its growth is accelerating. Big data applications deal with massive amounts of data. This field is growing quickly, creating lots of opportunities for software developers. Millions of information technology (IT) jobs globally already support big-data applications.

Twitter®—A Favorite Big-Data Source

One big-data source favored by developers is Twitter. There are approximately 800,000,000 tweets per day. Though tweets appear to be limited to 280 characters, Twitter actually provides almost 10,000 bytes of data per tweet to programmers who want to analyze tweets. So 800,000,000 times 10,000 is about 8,000,000,000,000 bytes or 8 terabytes (TB) of data per day. That's big data.

Prediction is a challenging and often costly process, but the potential rewards for accurate predictions are great. Data mining is the process of searching through extensive collections of data, often big data, to find insights that can be valuable to individuals and organizations. The sentiment that you data-mine from tweets could help predict the election results, the revenues a new movie is likely to generate and the success of a company's marketing campaign. It could also help companies spot weaknesses in competitors' product offerings.

/

Self Check

I (Fill-In) A(n) _____ is short for "binary digit"—a digit that can assume one of two values and is a computer's smallest data item.

Answer: bit.

^{19. &}quot;Twitter Usage Statistics." Accessed November 1, 2020. https://www.internetlivestats.com/twitter-statistics/.

2 (*TruelFalse*) In some operating systems, a file is viewed simply as a sequence of bytes—any organization of the bytes in a file, such as organizing the data into records, is a view created by the application programmer.

Answer: True.

3 (*Fill-In*) A database is a collection of data organized for easy access and manipulation. The most popular model is the _____ database, in which data is stored in simple tables.

Answer: relational.

1.4 Machine Languages, Assembly Languages and High-Level Languages

Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate translation steps. Hundreds of such languages are in use today. These may be divided into three general types:

- Machine languages.
- Assembly languages.
- High-level languages.

Machine Languages

Any computer can directly understand only its own machine language, defined by its hardware design. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are machine-dependent—a particular machine language can be used on only one type of computer. Such languages are cumbersome for humans. For example, here's a section of an early machine-language payroll program that adds overtime pay to base pay and stores the result in gross pay:

- +1300042774
- +1400593419
- +1200274027

In our Building Your Own Computer case study (Exercises 7.28–7.30), you'll "peel open" a computer and look at its internal structure. We'll introduce machine-language programming, and you'll write several machine-language programs. To make this an especially valuable experience, you'll then build a software simulation of a computer on which you can execute your machine-language programs.

Assembly Languages and Assemblers

Programming in machine language was simply too slow and tedious for most programmers. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations. These abbreviations formed the basis of assembly languages.

Translator programs called assemblers were developed to convert assembly-language programs to machine language at computer speeds. The following section of an assembly-language payroll program also adds overtime pay to base pay and stores the result in gross pay:

load basepay
add overpay
store grosspay

Although such code is clearer to humans, it's incomprehensible to computers until it's translated to machine language.

High-Level Languages and Compilers

With the advent of assembly languages, the use of computers increased rapidly. However, programmers still had to use numerous instructions to accomplish even simple tasks. To speed the programming process, high-level languages were developed in which single statements could accomplish substantial tasks. A typical high-level-language program contains many statements, known as the program's source code.

Translator programs called compilers convert high-level-language source code into machine language. High-level languages allow you to write instructions that look almost like everyday English and contain common mathematical notations. A payroll program written in a high-level language might contain a single statement such as

```
grossPay = basePay + overTimePay
```

From the programmer's standpoint, high-level languages are preferable to machine and assembly languages. C is among the world's most widely used high-level programming languages.

In our Building Your Own Compiler case study (Exercises 12.24–12.27), you'll build a compiler that takes programs written in a high-level programming language and converts them to the Simpletron Machine Language that you learn in Exercise 7.28. Exercises 12.24–12.27 "tie" together the entire programming process. You'll write programs in a simple high-level language, compile the programs on the compiler you build, then run the programs on the Simpletron simulator you build in Exercise 7.29.

Interpreters

Compiling a large high-level language program into machine language can take considerable computer time. Interpreters execute high-level language programs directly. Interpreters avoid compilation delays, but your code runs slower than compiled programs. Some programming languages, such as Java²⁰ and Python²¹, use a clever mixture of compilation and interpretation to run programs.

^{20. &}quot;Java virtual machine." Accessed November 2, 2020. https://en.wikipedia.org/wiki/Java_virtual_machine#Bytecode_interpreter_and_just-in-time_compiler.

^{21. &}quot;An introduction to Python bytecode." Accessed November 1, 2020. https://opensource.com/article/18/4/introduction-python-bytecode.

Self Check

- I (Fill-In) Translator programs called _____ convert assembly-language programs to machine language at computer speeds.

 Answer: assemblers.
- **2** (*Fill-In*) _____ programs, developed to execute high-level-language programs directly, avoid compilation delays, although they run slower than compiled programs **Answer**: Interpreter.
- 3 (*True/False*) High-level languages allow you to write instructions that look almost like everyday English and contain commonly used mathematical notations. Answer: True.

1.5 Operating Systems

Operating systems are software that make using computers more convenient for users, software developers and system administrators. They provide services that allow applications to execute safely, efficiently and concurrently with one another. The software that contains the core operating-system components is called the kernel. Linux, Windows and macOS are popular desktop computer operating systems—you can use any of these with this book. Each is partially written in C. The most popular mobile operating systems used in smartphones and tablets are Google's Android and Apple's iOS.

Windows—A Proprietary Operating System

In the mid-1980s, Microsoft developed the Windows operating system, consisting of a graphical user interface built on top of DOS (Disk Operating System)—an enormously popular personal-computer operating system that users interacted with by typing commands. Windows 10 is Microsoft's latest operating system—it includes the Cortana personal assistant for voice interactions. Windows is a proprietary operating system—it's controlled by Microsoft exclusively. It is by far the world's most widely used desktop operating system.

Linux—An Open-Source Operating System

The Linux operating system is among the greatest successes of the open-source movement. Proprietary software for sale or lease dominated software's early years. With open source, individuals and companies contribute to developing, maintaining and evolving the software. Anyone can then use that software for their own purposes—normally at no charge, but subject to a variety of (typically generous) licensing requirements. Open-source code is often scrutinized by a much larger audience than proprietary software, so errors can get removed faster, making the software more robust. Open source increases productivity and has contributed to an explosion of innovation. You'll use various popular open-source libraries and tools throughout this book.

There are many organizations in the open-source community. Some key ones are:

 GitHub (provides tools for managing open-source projects—it has millions of them under development).

14 Chapter I Introduction to Computers and C

- The Apache Software Foundation (originally the creators of the Apache web server) now oversees 350+ open-source projects, including several big-data infrastructure technologies.
- The Eclipse Foundation (the Eclipse Integrated Development Environment helps programmers conveniently develop software).
- The Mozilla Foundation (creators of the Firefox web browser).
- OpenML (which focuses on open-source tools and data for machine learning.
- OpenAI (which does research on artificial intelligence and publishes opensource tools used in AI reinforcement-learning research).
- OpenCV (which focuses on open-source computer-vision tools that can be used across various operating systems and programming languages.
- Python Software Foundation (responsible for the Python programming language).

Rapid improvements to computing and communications, decreasing costs and open-source software have made it much easier and more economical to create software-based businesses now than just a decade ago. A great example is Facebook, which was launched from a college dorm room and built with open-source software.

The Linux kernel is the core of the most popular open-source, freely distributed, full-featured operating system. It's developed by a loosely organized team of volunteers and is popular in servers, personal computers and embedded systems (such as the computer systems at the heart of smartphones, smart TVs and automobile systems). Unlike Microsoft's Windows and Apple's macOS source code, the Linux source code is available to the public for examination and modification and is free to download and install. As a result, Linux users benefit from a huge community of developers actively debugging and improving the kernel, and from the ability to customize the operating system to meet specific needs.

Apple's macOS and Apple's iOS for iPhone® and iPad® Devices

Apple, founded in 1976 by Steve Jobs and Steve Wozniak, quickly became a leader in personal computing. In 1979, Jobs and several Apple employees visited Xerox PARC (Palo Alto Research Center) to learn about Xerox's desktop computer that featured a graphical user interface (GUI). That GUI served as the inspiration for the Apple Macintosh, launched in 1984.

The Objective-C programming language, created by Stepstone in the early 1980s, added object-oriented programming (OOP) capabilities to the C programming language. Steve Jobs left Apple in 1985 and founded NeXT Inc. In 1988, NeXT licensed Objective-C from Stepstone. NeXT developed an Objective-C compiler and libraries, which were used as the platform for the NeXTSTEP operating system's user interface and Interface Builder (for constructing graphical user interfaces).

Jobs returned to Apple in 1996 when they bought NeXT. Apple's macOS operating system is a descendant of NeXTSTEP. Apple has several other proprietary operating systems derived from macOS:

- iOS is used in iPhones.
- iPadOS is used in iPads.
- watchOS is used in Apple Watches.
- tvOS is used in Apple TV devices.

In 2014, Apple introduced its Swift programming language, which it open-sourced in 2015. The Apple app-development community has largely shifted from Objective-C to Swift. Swift-based apps can import Objective-C and C software components.²²

Google's Android

Android—the most widely used mobile and smartphone operating system—is based on the Linux kernel, the Java programming language and, now, the open-source Kotlin programming language. Android is open source and free. Though you can't develop Android apps purely in C, you can incorporate C code into Android apps.²³

According to idc. com, 84.8% of smartphones shipped in 2020 use Android, compared to 15.2% for Apple.²⁴ The Android operating system is used in numerous smartphones, e-reader devices, tablets, TVs, in-store touch-screen kiosks, cars, robots, multimedia players and more.

Billions of Computerized Devices

Billions of personal computers and an even larger number of mobile devices are now in use. The explosive growth of smartphones, tablets and other devices creates significant opportunities for mobile-app developers. The following table lists many computerized devices, each of which can be part of the Internet of Things (see Section 1.11).

Some computerized device	es	
Automobiles	Blu-ray Disc TM players	Building controls
Cable boxes	Desktop computers	Credit cards
CT scanners	GPS navigation systems	e-Readers
Game consoles	Lottery systems	Home appliances
Home security systems	MRIs	Medical devices
Mobile phones	Parking meters	Personal computers
Optical sensors	Printers	Robots
Point-of-sale terminals	Servers	Smartcards
Smart meters	Televisions	Smartphones
Tablets	TV set-top boxes	Thermostats
Transportation passes	ATMs	Vehicle diagnostic systems

^{22. &}quot;Imported C and Objective-C APIs." Accessed November 3, 2020. https://developer.ap-ple.com/documentation/swift/imported_c_and_objective-c_apis.

^{23. &}quot;Add C and C++ code to your project." Accessed November 3, 2020. https://developer.android.com/studio/projects/add-native-code.

^{24. &}quot;Smartphone Market Share." Accessed December 24, 2020. https://www.idc.com/promo/smartphone-market-share/os.



I (*Fill-In*) Windows is a(n) _____ operating system—it's controlled by Microsoft exclusively.

Answer: proprietary.

2 (*TruelFalse*) Proprietary code is often scrutinized by a much larger audience than open-source software, so errors often get removed faster.

Answer: False. Open-source code is often scrutinized by a much larger audience than proprietary software, so errors often get removed faster.

3 (*True/False*) iOS dominates the global smartphone market over Android. Answer: False. Android currently controls 84.8% of the smartphone market, but iOS apps earn almost twice as much revenue as Android apps.²⁵

1.6 The C Programming Language

C evolved from two earlier languages, BCPL²⁶ and B²⁷. BCPL was developed in 1967 by Martin Richards as a language for writing operating systems and compilers. Ken Thompson modeled many features in his B language after their counterparts in BCPL, and in 1970 he used B to create early versions of the UNIX operating system at Bell Laboratories.

The C language was evolved from B by Dennis Ritchie at Bell Laboratories and was originally implemented in 1972.²⁸ C initially became widely known as the development language of the UNIX operating system. Many of today's leading operating systems are written in C and/or C++. C is mostly hardware-independent—with careful design, it's possible to write C programs that are **portable** to most computers.

Built for Performance

C is widely used to develop systems that demand performance, such as operating systems, embedded systems, real-time systems and communications systems:

Application	Description
Operating systems	C's portability and performance make it desirable for implementing operating systems, such as Linux and portions of Microsoft's Windows and Google's Android. Apple's macOS is built in Objective-C, which was derived from C. We discussed some key popular desktop/notebook operating systems and mobile operating systems in Section 1.5.

^{25. &}quot;Global App Revenue Reached \$50 Billion in the First Half of 2020, Up 23% Year-Over-Year." Accessed November 1, 2020. https://sensortower.com/blog/app-revenue-and-down-loads-1h-2020.

^{26. &}quot;BCLP." Accessed November 1, 2020. https://en.wikipedia.org/wiki/BCPL.

^{27. &}quot;B (programming language)." Accessed November 1, 2020. https://en.wikipedia.org/wiki/B_(programming_language).

^{28. &}quot;C (programming language)." Accessed November 1, 2020. https://en.wikipedia.org/wiki/C_(programming_language).

Application	Description
Embedded systems	The vast majority of the microprocessors produced each year are embedded in devices other than general-purpose computers. These embedded systems include navigation systems, smart home appliances, home security systems, smartphones, tablets, robots, intelligent traffic intersections and more. C is one of the most popular programming languages for developing embedded systems, which typically need to run as fast as possible and conserve memory. For example, a car's antilock brakes must respond immediately to slow or stop the car without skidding; video-game controllers should respond instantaneously to prevent lag between the controller and the game action.
Real-time systems	Real-time systems are often used for "mission-critical" applications that require nearly instantaneous and predictable response times. Real-time systems need to work continuously. For example, an air-traffic-control system must continuously monitor planes' positions and velocities and report that information to air-traffic controllers without delay so they can alert the planes to change course if there's a possibility of a collision.
Communications systems	Communications systems need to route massive amounts of data to their destinations quickly to ensure that things such as audio and video are delivered smoothly and without delay.

By the late 1970s, C had evolved into what's now referred to as "traditional C." The publication in 1978 of Kernighan and Ritchie's book, *The C Programming Language*, drew wide attention to the language. This became one of the most successful computer-science books of all time.

Standardization

C's rapid expansion to various hardware platforms (that is, types of computer hardware) led to many similar but often incompatible C versions. This was a serious problem for programmers who needed to develop code for several platforms. It became clear that a standard C version was needed. In 1983, the American National Standards Committee on Computers and Information Processing (X3) created the X3J11 technical committee to "provide an unambiguous and machine-independent definition of the language." In 1989, the standard was approved in the United States through the American National Standards Institute (ANSI), then worldwide through the International Standards Organization (ISO). This version was simply called Standard C.

The CII and CI8 Standards

We discuss the latest C standard (referred to as C11), which was approved in 2011 and updated with bug fixes in 2018 (referred to as C18). C11 refined and expanded C's capabilities. We've integrated into the text and Appendix C (in easy-to-include-or-omit sections) many of the new features implemented in leading C compilers. The current C standard document is referred to as *ISO/IEC 9899:2018*. Copies may be ordered from

According to the C standard committee, the next C standard is likely to be released in 2022.²⁹

Because C is a hardware-independent, widely available language, C applications often can run with little or no modification on a wide range of computer systems.



Self Check

- I (Fill-In) C evolved from two previous languages, _____ and ____. Answer: BCPL, B.
- **2** (*True/False*) It's possible to write C programs that are portable to most computers. **Answer**: *True*.

1.7 The C Standard Library and Open-Source Libraries

C programs consist of pieces called **functions**. You can program all the functions you need to form a C program. However, most C programmers take advantage of the rich collection of existing functions in the C **standard library**. Thus, there are really two parts to learning C programming:

- learning the C language itself, and
- learning how to use the functions in the C standard library.

Throughout the book, we discuss many of these functions. P. J. Plauger's book *The Standard C Library* is must reading for programmers who need a deep understanding of the library functions, how to implement them and how to use them to write portable code. We use and explain many C library functions throughout this text.

C How to Program, 9/e encourages a building-block approach to creating programs. When programming in C, you'll typically use the following building blocks:

- C standard library functions,
- open-source C library functions,
- functions you create yourself, and
- functions other people (whom you trust) have created and made available to you.

The advantage of creating your own functions is that you'll know exactly how they work. The disadvantage is the time-consuming effort that goes into designing, developing, debugging and performance-tuning new functions. Throughout the book, we focus on using the existing C standard library to leverage your program-development efforts and avoid "reinventing the wheel." This is called **software reuse**.



Using C standard library functions instead of writing your own versions can improve program performance, because these functions are carefully written to perform efficiently. Using C standard library functions instead of writing your own comparable versions also can improve program portability.

^{29. &}quot;Programming Language C — C2x Charter." Accessed November 4, 2020. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2086.htm.

Open-Source Libraries

There are enormous numbers of third-party and open-source C libraries that can help you perform significant tasks with modest amounts of code. GitHub lists over 32,000 repositories in their C category:

https://github.com/topics/c

In addition, pages such as *Awesome C*

https://github.com/kozross/awesome-c

provide curated lists of popular C libraries for a wide range of application areas.

Self Check

(Fill-In) Most C programmers take advantage of the rich collection of existing functions called the __

Answer: C standard library.

2 (*Fill-In*) Avoid "reinventing the wheel" Instead, use existing pieces. This is called

Answer: software reuse.

1.8 Other Popular Programming Languages

The following is a brief intro to several other popular programming languages:

- BASIC was developed in the 1960s at Dartmouth College to familiarize novices with programming techniques. Many of its latest versions are object-oriented.
- C++, which is based on C, was developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories. C++ provides features that enhance the C language and adds object-oriented programming capabilities. We introduce object-oriented programming concepts in Appendix D.
- *Python* is an object-oriented language that was released publicly in 1991. It was developed by Guido van Rossum of the National Research Institute for Mathematics and Computer Science in Amsterdam. Python has rapidly become one of the world's most popular programming languages, especially for educational and scientific computing, and in 2017 it surpassed the programming language R as the most popular data-science programming language. 30,31,32 Some rea-

^{30. &}quot;5 things to watch in Python in 2017." Accessed November 1, 2020. https://www.oreilly.com/ideas/5-things-to-watch-in-python-in-2017.

^{31. &}quot;Python overtakes R, becomes the leader in Data Science, Machine Learning platforms." Accessed November 1, 2020. https://www.kdnuggets.com/2017/08/python-overtakes-r-leaderanalytics-data-science.html.

^{32. &}quot;Data Science Job Report 2017: R Passes SAS, But Python Leaves Them Both Behind." Accessed November 1, 2020. https://www.r-bloggers.com/data-science-job-report-2017-rpasses-sas-but-python-leaves-them-both-behind/.

sons why Python is popular: ^{33,34,35} It's open-source, free and widely available. It's supported by a massive open-source community. It's relatively easy to learn. Its code is easier to read than many other popular programming languages. It enhances developer productivity with extensive standard libraries and thousands of third-party open-source libraries. It's popular in web development and in artificial intelligence, which is enjoying explosive growth, in part because of its special relationship with data science. It's widely used in the financial community. ³⁶

- Java—Sun Microsystems in 1991 funded an internal corporate research project led by James Gosling, which resulted in the C++-based object-oriented programming language called Java. A key Java goal is "write once, run anywhere," enabling developers to write programs that run on a wide variety of computer systems. Java is used in enterprise applications, in web servers (the computers that provide the content to our web browsers), in applications for consumer devices (e.g., smartphones, tablets, television set-top boxes, appliances, automobiles and more) and for many other purposes. Java was originally the preferred Android app-development language, though several other languages are now supported.
- *C#* (based on C++ and Java) is one of Microsoft's three primary object-oriented programming languages—the other two are Visual C++ and Visual Basic. C# was developed to integrate the web into computer applications and is now widely used to develop many kinds of applications. As part of Microsoft's many open-source initiatives, they now offer open-source versions of C# and Visual Basic.
- JavaScript is a widely used scripting language that's primarily used to add programmability to web pages (e.g., animations, user interactivity and more). All major web browsers support it. Many Python visualization libraries output JavaScript to create interactive visualizations you can view in your web browser. Tools such as NodeJS also enable JavaScript to run outside of web browsers.
- Swift, which was introduced in 2014, is Apple's programming language for developing iOS and macOS apps. Swift is a contemporary language that includes popular features from Objective-C, Java, C#, Ruby, Python and other languages. Swift is open-source, so it can be used on non-Apple platforms as well.

^{33. &}quot;Why Learn Python? Here Are 8 Data-Driven Reasons." Accessed November 1, 2020. https://dbader.org/blog/why-learn-python.

^{34. &}quot;Why Learn Python." Accessed November 1, 2020. https://simpleprogrammer.com/7-reasons-why-you-should-learn-python.

^{35. &}quot;5 things to watch in Python in 2017." Accessed November 1, 2020. https://www.oreil-ly.com/ideas/5-things-to-watch-in-python-in-2017.

^{36.} Kolanovic, M. and R. Krishnamachari, Big Data and AI Strategies: Machine Learning and Alternative Data Approach to Investing (J.P. Morgan, 2017).

 R is a popular open-source programming language for statistical applications and visualization. Python and R are the two most widely used data-science languages.

✓ Self Check

- **I** (*Fill-In*) Today, most code for general-purpose operating systems and other performance-critical systems is written in _____. **Answer:** C or C++.
- **2** (*Fill-In*) A key goal of ______ is "write once, run anywhere," enabling developers to write programs that will run on a great variety of computer systems and computer-controlled devices. **Answer:** Java.
- **3** (*True/False*) R is the most popular data-science programming language. **Answer:** False. In 2017, Python surpassed R as the most popular data-science programming language.

1.9 Typical C Program-Development Environment

C systems generally consist of several parts: a program-development environment, the language and the C standard library. The following discussion explains the typical C development environment.

C programs typically go through six phases to be executed—edit, preprocess, compile, link, load and execute. Although *C How to Program*, *9/e*, is a generic C textbook (written independently of any particular operating system), we concentrate in this section on a typical Linux-based C system. In Section 1.10, you'll test-drive creating and running C programs on Windows, macOS and/or Linux.

1.9.1 Phase 1: Creating a Program

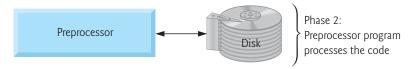
Phase 1 (in the following diagram) consists of editing a file in an editor program:



Two editors widely used on Linux systems are vi and emacs. C and C++ integrated development environments (IDEs) such as Microsoft Visual Studio and Apple Xcode have integrated editors. You type a C program in the editor, make corrections if necessary, then store the program on a secondary storage device such as a hard disk. C program filenames should end with the .c extension.

1.9.2 Phases 2 and 3: Preprocessing and Compiling a C Program

In Phase 2 (shown in the following diagram), you give the command to compile the program:



The compiler translates the C program into machine-language code (also referred to as object code). In a C system, the compilation command invokes a preprocessor program before the compiler's translation phase begins. The C preprocessor obeys special commands called preprocessor directives, which perform text manipulations on a program's source-code files. These manipulations consist of inserting the contents of other files and various text replacements. The early chapters discuss the most common preprocessor directives. Chapter 14 discusses other preprocessor features.

In Phase 3 (shown in the following diagram), the compiler translates the C program into machine-language code:



A syntax error occurs when the compiler cannot recognize a statement because it violates the language rules. The compiler issues an error message to help you locate and fix the incorrect statement. The C standard does not specify the wording for error messages issued by the compiler, so the messages you see on your system may differ from those on other systems. Syntax errors are also called **compile errors** or **compile-time errors**.

1.9.3 Phase 4: Linking

The next phase (shown in the following diagram) is called linking:

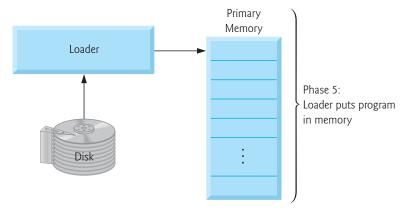


C programs typically use functions defined elsewhere, such as in the standard libraries, open-source libraries or private libraries of a particular project. The object code produced by the C compiler typically contains "holes" due to these missing parts. A linker links a program's object code with the code for the missing functions to produce an executable image (with no missing pieces). On a typical Linux system, the command to compile and link a program is gcc (the GNU C compiler). To compile and link a program named welcome.c using the latest C standard (C18), type

at the Linux prompt and press the *Enter* key (or *Return* key). Linux commands are case sensitive. If the program compiles and links correctly, the compiler produces a file named a out (by default), which is we1come.c's executable image.

1.9.4 Phase 5: Loading

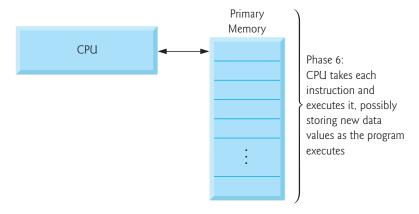
The next phase (shown in the following diagram) is called loading:



Before a program can execute, the operating system must load it into memory. The loader takes the executable image from disk and transfers it to memory. Additional components from shared libraries that support the program also are loaded.

1.9.5 Phase 6: Execution

Finally, in the last phase (shown in the following diagram), the computer, under the control of its CPU, executes the program one instruction at a time:



To load and execute the program on a Linux system, type ./a.out at the Linux prompt and press *Enter*.

1.9.6 Problems That May Occur at Execution Time

Programs do not always work on the first try. Each of the preceding phases can fail because of various errors that we'll discuss. For example, an executing program might attempt to divide by zero (an illegal operation on computers just as in arithmetic). This would cause the computer to display an error message. You would then return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine that the corrections work properly.

ERR ⊗

Errors such as division-by-zero that occur as programs run are called runtime errors or execution-time errors. Divide-by-zero is generally a fatal error that causes the program to terminate immediately without successfully performing its job. Nonfatal errors allow programs to run to completion, often producing incorrect results.

1.9.7 Standard Input, Standard Output and Standard Error Streams

Most C programs input and/or output data. Certain C functions take their input from **stdin** (the **standard input stream**), which is normally the keyboard. Data is often output to **stdout** (the **standard output stream**), which is normally the computer screen. When we say that a program prints a result, we normally mean that the result is displayed on a screen. Data also may be output to devices such as disks and printers. There's also a **standard error stream** referred to as **stderr**, which is normally connected to the screen and used to display error messages. It's common to route regular output data, i.e., **stdout**, to a device other than the screen while keeping **stderr** assigned to the screen so that the user can be immediately informed of errors.



Self Check

I (Fill-In) C programs typically go through six phases to be executed:
Answer: edit, preprocess, compile, link, load, execute.
2 (Fill-In) A(n) occurs when the compiler cannot recognize a statemen because it violates the rules of the language.
Answer: syntax error.
3 (Fill-In) Errors that occur as a program runs are called or execution
time errors
Answer: runtime errors.

1.10 Test-Driving a C Application in Windows, Linux and macOS

In this section, you'll compile, run and interact with your first C application—a guess-the-number game, which picks a random number from 1 to 1000 and prompts you to guess it. If you guess correctly, the game ends. If you guess incorrectly, the application indicates whether your guess is higher or lower than the correct number. There's no limit to your number of guesses, but you should be able to guess a number from 1 to 1000 correctly in 10 or fewer tries. There's some nice computer science behind this game—in a later chapter, you'll explore the binary search technique

You'll create this application in Chapter 5's exercises. Usually, this application randomly selects the correct answers. We disabled random selection for the test-drives. The application uses the same correct answer every time you run it. That way, you can use the same guesses we use and see the same results. This answer may vary by compiler.

Summary of the Test-Drives

We'll demonstrate creating a C application using:

- Microsoft Visual Studio 2019 Community edition for Windows (Section 1.10.1).
- Clang in Xcode on macOS (Section 1.10.2).
- GNU gcc in a shell on Linux (Section 1.10.3).
- GNU gcc in a shell running inside the GNU Compiler Collection (GCC) Docker container (Section 1.10.4).

You need to read only the section that corresponds to your setup.

Many development environments are available in which you can compile, build and run C applications. If your course uses a different tool from those we demonstrate here, consult your instructor for information on that tool.

1.10.1 Compiling and Running a C Application with Visual Studio 2019 Community Edition on Windows 10

In this section, you'll run a C program on Windows using Microsoft Visual Studio 2019 Community edition. Several versions of Visual Studio are available. In some versions, the options, menus and instructions we present might differ slightly. From this point forward, we'll simply say "Visual Studio" or "the IDE."

Step 1: Checking Your Setup

If you have not already done so, read the Before You Begin section of this book for instructions on installing the IDE and downloading the book's code examples.

Step 2: Launching Visual Studio

Launch Visual Studio from the **Start** menu. Dismiss the initial Visual Studio window by pressing the *Esc* key. **Do not** click the **X** in the upper-right corner, as that will terminate Visual Studio. You can access this window at any time by selecting **File > Start Window**. We use > to indicate selecting a menu item from a menu, so **File > Open** means "select the **Open** menu item from the **File** menu."

Step 3: Creating a Project

A project is a group of related files, such as the C source-code files that compose an application. Visual Studio organizes applications into projects and solutions, which contain one or more projects. Programmers use multiple-project solutions to create large-scale applications. Our examples require only single-project solutions. For our code examples, you'll begin with an Empty Project and add files to it. To create a project:

- 1. Select File > New > Project... to display the Create a new project dialog.
- 2. Select the Empty Project template with the tags C++, Windows and Console. Visual Studio does not have a C compiler, but its Visual C++ compiler can compile most C programs. The template we use here is for programs that execute at the command line in a Command Prompt window. Depending on

your Visual Studio version and the installed options, there may be many other project templates. You can filter your choices using the Search for templates textbox and the drop-down lists below it. Click Next to display the Configure your new project dialog.

3. Provide a Project name and Location. For the Project name, we specified c_test. For the Location, we selected this book's examples folder, which we assume is in your user account's Documents folder. Click Create to open your new project in Visual Studio.

At this point, Visual Studio creates your project, places its folder in

C:\Users\YourUserAccount\Documents\examples

(or the folder you specified) and opens Visual Studio's main window.

When you edit C code, Visual Studio displays each file as a separate tab within the window. The **Solution Explorer**—docked to Visual Studio's left or right side—is for viewing and managing your application's files. In this book's examples, you'll typically place each program's code files in the **Source Files** folder. If the **Solution Explorer** is not displayed, you can display it by selecting **View > Solution Explorer**.

Step 4: Adding the GuessNumber.c File into the Project

Next, let's add the file GuessNumber.c to the project. In the Solution Explorer:

- 1. Right-click the Source Files folder and select Add > Existing Item....
- 2. In the dialog that appears, navigate to the ch01 subfolder of the book's examples folder, select GuessNumber.c and click Add.³⁷

Step 5: Configuring Your Project's Compiler Version and Disabling a Microsoft Error Message

The Before You Begin section mentioned that Visual C++ can compile most C programs. The Visual C++ compiler supports several C++ standard versions. We'll use Microsoft's C++17 compiler, which we must configure in our project's settings:

- 1. Right-click the project's node— **t** c_test —in the Solution Explorer and select Properties to display the project's C_test Property Pages dialog.
- 2. In the Configuration drop-down list, change Active(Debug) to All Configurations. In the Platform drop-down list, change Active(Win32) to All Platforms.
- 3. In the left column, expand the C/C++ node, then select Language.
- **4.** In the right column, click in the field to the right of C++ Language Standard, click the down arrow, then select ISO C++17 Standard (/std:c++17).³⁸

^{37.} For the multiple-source-code-file programs that you'll see in later chapters, select all the files for a given program. When you begin creating programs yourself, you can right-click the Source Files folder and select Add > New Item... to display a dialog for adding a new file. You'll need to change the filename extension from .cpp to .c for your C program files.

^{38.} At the time of this writing, Microsoft was still completing its support for the C++20 standard. Once available, you should choose ISO C++20 Standard (/std:c++20).

- 5. In the left column, in the C/C++ node, select Preprocessor.
- **6.** In the right column, at the end of the value for **Preprocessor Definitions**, insert :_CRT_SECURE_NO_WARNINGS
- 7. In the left column, in the C/C++ node, select General.
- **8.** In the right column, click in the field to the right of **SDL** checks, click the down arrow, then select **No** (/sdl-).
- **9.** Click **OK** to save the changes.

Items 6 and 8 above eliminate Microsoft Visual C++ warning and error messages for several C library functions we use throughout this book. We'll say more about this issue in Section 3.13.

Step 6: Compiling and Running the Project

Next, let's compile and run the project so you can test-drive the application. Select **Debug > Start without debugging** or type Ctrl + F5. If the program compiles correctly, Visual Studio opens a Command Prompt window and executes the program. We changed the Command Prompt's color scheme³⁹ and font size for readability:

```
I have a number between 1 and 1000.

Can you guess my number?

Please type your first guess.

?
```

Step 7: Entering Your First Guess

At the ? prompt, type 500 and press *Enter*. The application displays, "Too high. Try again." to indicate that the value you entered is greater than the number the application chose as the correct guess:

```
Can you guess my number?

Can you guess my number?

Please type your first guess.

? 500

Too high. Try again.

?
```

Step 8: Entering Another Guess

At the next prompt, type **250** and press *Enter*. The application displays, "Too high. Try again." to indicate that the value you entered is greater than the correct guess:

^{39.} If you'd like to modify the Command Prompt colors on your system, right click the title bar and select Properties. In the "Command Prompt" Properties dialog, click the Colors tab, and select your preferred text and background colors.

```
I have a number between 1 and 1000.

Can you guess my number?

Please type your first guess.

? 500

Too high. Try again.

? 250

Too high. Try again.

? "
```

Step 9: Entering Additional Guesses

Continue to play the game by entering values until you guess the correct number. When you guess correctly, the application displays, "Excellent! You guessed the number!":

```
C\Users\PaulDeitel\Documents\examples\cpp20_test\Debug\c_test.exe
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too high. Try again.
? 125
Too high. Try again.
? 63
Too high. Try again.
Too low. Try again.
Too high. Try again.
? 39
Too low. Try again.
? 43
Too high. Try again.
Too low. Try again.
? 42
Excellent! You guessed the number!
Would you like to play again?
Please type (1=yes, 2=no)? _
```

Step 10: Playing the Game Again or Exiting the Application

After you guess the correct number, the application asks if you'd like to play another game. At the "Please type (1=yes, 2=no)?" prompt, enter 1 to play again, which chooses a new number to guess. Enter 2 if you wish to terminate the application. Each time you execute this application (*Step 6*), it will choose the same numbers for you to guess. To play a randomized version of the game, use the version of GuessNumber.c in the ch01 folder's randomized_version subfolder.

Reusing This Project for Subsequent Examples

You can follow the steps in this section to create a separate project for every application in the book. However, for our examples, you may find it more convenient to reuse this project by removing the GuessNumber.c program from the project, then adding another C program. To remove a file from your project (but not your system), select it in the Solution Explorer, then press *Del* (or *Delete*). Repeat *Step 4* to add a different program to the project.

Using Ubuntu Linux in the Windows Subsystem for Linux

Some Windows users may want to use the GNU gcc compiler on Windows, especially for the few programs in this book that Visual C++ cannot compile. You can do this using the GNU Compiler Collection Docker container (Section 1.10.4), or you can use gcc in Ubuntu Linux running in the Windows Subsystem for Linux. To install the Windows Subsystem for Linux, follow the instructions at

https://docs.microsoft.com/en-us/windows/wsl/install-win10

Once you install and launch Ubuntu on your Windows System, you can use the following command to change to the folder containing the test-drive code example on your Windows system:

cd /mnt/c/Users/*YourUseName*/Documents/examples/ch01

Then you can continue with *Step 2* in Section 1.10.3.

1.10.2 Compiling and Running a C Application with Xcode on macOS

In this section, you'll run a C program on a macOS using the Clang compiler in Apple's Xcode IDE.

Step 1: Checking Your Setup

If you have not already done so, read the Before You Begin section of this book for instructions on installing the IDE and downloading the book's code examples.

Step 2: Launching Xcode

Open a Finder window, select **Applications** and double-click the Xcode icon (). If this is your first time running Xcode, the **Welcome to Xcode** window appears. Close this window by clicking the X in the upper-left corner—you can access it at any time by selecting **Window > Welcome to Xcode**. We use the > character to indicate selecting a menu item from a menu. For example, the notation **File > Open...** means "select the **Open...** menu item from the **File** menu."

Step 3: Creating a Project

A project is a group of related files, such as the C source-code files that compose an application. The Xcode projects we created for this book's examples are Command Line Tool projects that you'll execute in the IDE. To create a project:

1. Select File > New > Project....

- 2. At the top of the Choose a template for your new project dialog, click macOS.
- 3. Under Application, click Command Line Tool and click Next.
- **4.** For **Product Name**, enter a name for your project—we specified C_test_Xcode.
- 5. In the Language drop-down list, select C, then click Next.
- **6.** Specify where you want to save your project. We selected the examples folder containing this book's code examples.
- Click Create.

Xcode creates your project and displays the workspace window, initially showing three areas—the **Navigator** area (left), **Editor** area (middle) and **Utilities** area (right).

The left-side **Navigator** area has icons at its top for the *navigators* that can be displayed there. For this book, you'll primarily work with

- **Project** ()—Shows all the files and folders in your project.
- **Issue** (⚠)—Shows you warnings and errors generated by the compiler.

Clicking a navigator button displays the corresponding navigator panel.

The middle Editor area is for managing project settings and editing source code. This area is always displayed in your workspace window. Selecting a file in the Project navigator displays the file's contents in the Editor area. You will not use the right-side Utilities area in this book. You'll run and interact with the guess-the-number program in the Debug area, which will appear below the Editor area.

The workspace window's toolbar contains options for executing a program, displaying the progress of tasks executing in Xcode and hiding or showing the left (Navigator), right (Utilities) and bottom (Debug) areas.

Step 4: Deleting the main.c File from the Project

By default, Xcode creates a main.c source-code file containing a simple program that displays, "Hello, World!". You won't use main.c in this test-drive. In the Project navigator, right-click the main.c file and select Delete. In the dialog that appears, select Move to Trash. The file will be removed from your system if you empty your trash.

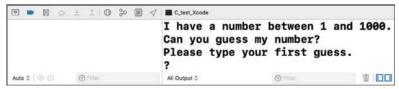
Step 5: Adding the GuessNumber.c File into the Project

In a Finder window, open the ch01 folder in the book's examples folder, then drag GuessNumber.c onto the Project navigator's C_Test_Xcode folder. In the dialog that appears, ensure that Copy items if needed is checked, then click Finish.⁴⁰

Step 6: Compiling and Running the Project

To compile and run the project so you can test-drive the application, simply click the run () button on Xcode's toolbar. If the program compiles correctly, Xcode opens the **Debug** area and executes the program in the right half of the **Debug** area:

^{40.} For the multiple-source-code-file programs that you'll see later in the book, drag all the files for a given program to the project's folder. When you begin creating your own programs, you can right-click the project's folder and select New File... to display a dialog for adding a new file.



The application displays, "Please type your first guess.", then displays a question mark (?) as a prompt on the next line.

Step 7: Entering Your First Guess

Click in the **Debug** area, then type **500** and press *Return*:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too low. Try again.
? |
```

The application displays, "Too low. Try again.", meaning that the value you entered is less than the number the application chose as the correct guess.

Step 8: Entering Another Guess

At the next prompt, enter **750**:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too low. Try again.
? 750
Too low. Try again.
?
```

The application displays, "Too low. Try again.", meaning the value you entered once again is less than the correct guess.

Step 9: Entering Additional Guesses

Continue to play the game until you guess the correct number. When you guess correctly, the application displays, "Excellent! You guessed the number!":

```
? 875
Too high. Try again.
? 812
Too high. Try again.
? 781
Too low. Try again.
? 797
Too low. Try again.
? 805
Too low. Try again.
? 808

Excellent! You guessed the number!
Would you like to play again?
Please type (1=yes, 2=no)?
```

Playing the Game Again or Exiting the Application

After you guess the correct number, the application asks if you'd like to play another game. At the "Please type (1=yes, 2=no)?" prompt, enter 1 to play again, which chooses a new number to guess. Enter 2 if you wish to terminate the application. Each time you execute this application (*Step 6*), it will choose the same numbers for you to guess. To play a randomized version of the game, use the version of GuessNumber.c in the ch01 folder's randomized_version subfolder.

Reusing This Project for Subsequent Examples

You can follow the steps in this section to create a separate project for every application in the book. For our examples, you may find it more convenient to reuse this project by removing the project's current program, then adding a new one. To remove a file from your project (but not your system), right-click the file in the **Project** navigator and select **Delete**. In the dialog that appears, select **Remove Reference**. You can then repeat *Step 6* to add a different program to the project.

1.10.3 Compiling and Running a C Application with GNU gcc on Linux

For this test-drive, we assume that you read the Before You Begin section and that you placed the downloaded examples in your user account's Documents folder.

Step 1: Changing to the ch01 Folder

From a Linux shell, use the cd command to change to the ch01 subfolder of the book's examples folder:

```
~$ cd ~/Documents/examples/ch01
~/Documents/examples/ch01$
```

In this section's figures, we use **bold** to highlight the text you should type. The prompt in our Ubuntu Linux shell uses a tilde (~) to represent your home directory. Each prompt ends with a dollar sign (\$). The prompt may differ on other Linux distributions.

Step 2: Compiling the Application

Before running the application, you must first compile it:

```
~/Documents/examples/ch01$ gcc -std=c18 GuessNumber.c -o GuessNumber ~/Documents/examples/ch01$
```

The gcc command compiles the application:

- The -std=c18 option indicates that we're using C18—the latest version of the C programming language standard.
- The -o option names the executable file (GuessNumber) you'll use to run the program.

Step 3: Running the Application

Type ./GuessNumber at the prompt and press *Enter* to run the program:

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

The ./ tells Linux to run a file from the current directory. It's needed here to indicate that GuessNumber is an executable file.

Step 4: Entering Your First Guess

The application displays, "Please type your first guess.", then displays a question mark (?) as a prompt on the next line. At the prompt, enter 500—note that the outputs may vary based on the compiler you're using:

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

The application displays, "Too high. Try again.", meaning the value you entered is greater than the number the application chose as the correct guess.

Step 5: Entering Another Guess

At the next prompt, enter **250**:

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too low. Try again.
?
```

This time the application displays, "Too Tow. Try again.", meaning the value you entered is less than the correct guess.

Step 6: Entering Additional Guesses

Continue to play the game by entering values until you guess the correct number. When you guess correctly, the application displays, "Excellent! You guessed the number!":

```
Too low. Try again.
? 375
Too low. Try again.
? 437
Too high. Try again.
? 406
Too high. Try again.
? 391
Too high. Try again.
? 383
Too low. Try again.
? 387
Too high. Try again.
? 385
Too high. Try again.
? 384
Excellent! You guessed the number!
Would you like to play again?
Please type (1=yes, 2=no)?
```

Step 7: Playing the Game Again or Exiting the Application

After you guess the correct number, the application asks if you'd like to play another game. At the "Please type (1=yes, 2=no)?" prompt, enter 1 to play again, which chooses a new number to guess. Enter 2 if you wish to terminate the application. Each time you execute this application (*Step 3*), it will choose the same numbers for you to guess. To play a randomized version of the game, use the version of GuessNumber.c in the ch01 folder's randomized version subfolder.

1.10.4 Compiling and Running a C Application in a GCC Docker Container Running Natively over Windows 10, macOS or Linux

One of the most convenient cross-platform ways to run GNU's gcc compiler is via the GNU Compiler Collection (GCC) Docker container. This section assumes you've installed Docker Desktop (Windows or macOS) or Docker Engine (Linux)—as discussed in the Before You Begin section—and that Docker is running on your computer.

Executing the GNU Compiler Collection (GCC) Docker Container

Open a Command Prompt (Windows), Terminal (macOS/Linux) or shell (Linux), then perform the following steps to launch the GCC Docker container:

- 1. Use the cd command to navigate to the examples folder containing this book's examples. Executing the Docker container from here will enable the container to access our code examples.
- 2. Windows users: Launch the GCC Docker container with the command⁴¹ docker run --rm -it -v "%CD%":/usr/src gcc:latest

^{41.} A notification will appear asking you to allow Docker to access the files in the current folder. You must allow this; otherwise, you will not be able to access our source-code files in Docker.

3. macOS/Linux users: Launch the GCC Docker container with the command docker run --rm -it -v "\$(pwd)":/usr/src gcc:latest

In the preceding commands:

- --rm cleans up the GCC container's resources when you eventually shut it down.
- -it runs the container in interactive mode, so you can enter commands to change folders, compile programs using the GNU gcc compiler and run programs.
- -v "%CD%":/usr/src (Windows) or -v "\$(pwd)":/usr/src (macOS/Linux) allows the Docker container to access your current folder's files and subfolders via the Docker container's /usr/src folder. You can navigate with the cd command to subfolders of /usr/src to compile and run our programs.
- gcc:latest is the container name that you installed in the Before You Begin.⁴²

Once the container is running, you'll see a prompt similar to

```
root@67773f59d9ea:/#
```

though "@67773f59d9ea" will differ on your computer. The container uses a Linux operating system in which folder separators are forward slashes (/). The prompt displays the current folder location between the : and #.

Changing to the ch01 Folder in the Docker Container

Use the cd command to change to the /usr/src/ch01 folder:

```
root@01b4d47cadc6:/# cd /usr/src/ch01
root@01b4d47cadc6:/usr/src/ch01#
```

You can now compile, run and interact with the GuessNumber application in the Docker container, using the commands in Section 1.10.3, *Steps 2–7*.

Terminating the Docker Container

You can terminate the Docker container by typing Ctrl + d at the container's prompt.

1.11 Internet, World Wide Web, the Cloud and IoT

In the late 1960s, ARPA—the Advanced Research Projects Agency of the United States Department of Defense—rolled out plans for networking the main computer systems of approximately a dozen ARPA-funded universities and research institu-

^{42.} gcc:latest is the name of the gcc Docker container's latest version at the time you downloaded it onto your machine. Once downloaded, the container does not auto-update. You can keep your GCC container up-to-date with the latest available release by executing docker pull gcc:latest. If there's a new version, Docker will download it.

tions. The computers were to be connected with communications lines operating at speeds on the order of 50,000 bits per second, a stunning rate at a time when most people (of the few who even had networking access) were connecting over telephone lines to computers at a rate of 110 bits per second. Academic research was about to take a giant leap forward. ARPA proceeded to implement what quickly became known as the ARPANET, the precursor to today's Internet. Today's fastest Internet speeds are on the order of billions of bits per second, with trillion-bits-per-second (terabit) speeds already being tested! In 2020, Australian researchers successfully tested a 44.2 terrabits per second Internet connection.

Things worked out differently from the original plan. Although the ARPANET enabled researchers to network their computers, its main benefit proved to be the capability for quick and easy communication via what came to be known as electronic mail (e-mail). This is true even on today's Internet, with e-mail, instant messaging, file transfer and social media, such as Snapchat, Instagram, Facebook and Twitter, enabling billions of people worldwide to communicate quickly and easily.

The protocol (set of rules) for communicating over the ARPANET became known as the Transmission Control Protocol (TCP). TCP ensured that messages, consisting of sequentially numbered pieces called packets, were properly delivered from sender to receiver, arrived intact and were assembled in the correct order.

1.11.1 The Internet: A Network of Networks

In parallel with the early evolution of the Internet, organizations worldwide were implementing their own networks for intra-organization (that is, within an organization) and inter-organization (that is, between organizations) communication. A huge variety of networking hardware and software appeared. One challenge was to enable these different networks to communicate with each other. ARPA accomplished this by developing the Internet Protocol (IP), which created a true "network of networks," the Internet's current architecture. The combined set of protocols is now called TCP/IP. Each Internet-connected device has an IP address—a unique numerical identifier used by devices communicating via TCP/IP to locate one another on the Internet.

Businesses rapidly realized that, by using the Internet, they could improve their operations and offer new and better services to their clients. Companies started spending large amounts of money to develop and enhance their Internet presence. This generated fierce competition among communications carriers and hardware and software suppliers to meet the increased infrastructure demand. As a result, Internet bandwidth—the information-carrying capacity of communications lines—has increased tremendously, while hardware costs have plummeted.

^{43. &}quot;BT Testing 1.4 Terabit Internet Connections." Accessed November 1, 2020. https://testinternetspeed.org/blog/bt-testing-1-4-terabit-internet-connections/.

^{44. &}quot;Monash, Swinburne, and RMIT universities use optical chip to achieve 44Tbps data speed." Accessed January 9, 2021. https://www.zdnet.com/article/monash-swinburne-and-rmit-universities-achieve-44tbps-data-speed-using-single-optical-chip/.

1.11.2 The World Wide Web: Making the Internet User-Friendly

The World Wide Web (simply called "the web") is a collection of hardware and software associated with the Internet that allows computer users to locate and view documents (with various combinations of text, graphics, animations, audios and videos) on almost any subject. In 1989, Tim Berners-Lee of CERN (the European Organization for Nuclear Research) began developing HyperText Markup Language (HTML)—the technology for sharing information via "hyperlinked" text documents. He also wrote communication protocols such as HyperText Transfer Protocol (HTTP) to form the backbone of his new hypertext information system, which he referred to as the World Wide Web.

In 1994, Berners-Lee founded the World Wide Web Consortium (W3C, https://www.w3.org), devoted to developing web technologies. One of the W3C's primary goals is to make the web universally accessible to everyone regardless of disabilities, language or culture.

1.11.3 The Cloud

More and more computing today is done "in the cloud"—that is, using software and data distributed across the Internet worldwide, rather than locally on your desktop, notebook computer or mobile device. Cloud computing allows you to increase or decrease computing resources to meet your needs at any given time, which is more cost-effective than purchasing hardware to provide enough storage and processing power to meet occasional peak demands. Cloud computing also saves money by shifting to the service provider the burden of managing these apps (such as installing and upgrading the software, security, backups and disaster recovery).

The apps you use daily are heavily dependent on various cloud-based services. These services use massive clusters of computing resources (computers, processors, memory, disk drives, etc.) and databases that communicate over the Internet with each other and the apps you use. A service that provides access to itself over the Internet is known as a web service.

Software as a Service

Cloud vendors focus on service-oriented architecture (SOA) technology. They provide "as-a-Service" capabilities that applications connect to and use in the cloud. Common services provided by cloud vendors include:⁴⁵

"As-a-Service" acronyms (note that several are the same)					
Big data as a Service (BDaaS)	Platform as a Service (PaaS)				
Hadoop as a Service (HaaS)	Software as a Service (SaaS)				
Infrastructure as a Service (IaaS)	Storage as a Service (SaaS)				

^{45.} For more "as-a-Service" acronyms, see https://en.wikipedia.org/wiki/Cloud_computing and https://en.wikipedia.org/wiki/As_a_service.

Mashups

The applications-development methodology of mashups enables you to rapidly develop powerful software applications by combining (often free) complementary web services and other forms of information feeds. One of the first mashups, www.housingmaps.com, combined the real-estate listings from www.craigslist.org with Google Maps to show the locations of homes for sale or rent in a given area. Check out www.housingmaps.com for some interesting facts, history, articles and how it influenced real-estate industry listings.

ProgrammableWeb (https://programmableweb.com/) provides a directory of nearly 24,000 web services and almost 8,000 mashups. They also provide how-to guides and sample code for working with web services and creating your own mashups. According to their website, some of the most widely used web services are Google Maps and others provided by Facebook, Twitter and YouTube.

1.11.4 The Internet of Things

The Internet is no longer just a network of *computers*—it's an Internet of Things (IoT). A *thing* is any object with an IP address and the ability to send, and in some cases receive, data automatically over the Internet. Such *things* include:

- a car with a transponder for paying tolls,
- monitors for parking-space availability in a garage,
- a heart monitor implanted in a human,
- water-quality monitors,
- a smart meter that reports energy usage,
- radiation detectors,
- item trackers in a warehouse,
- mobile apps that can track your movement and location,
- smart thermostats that adjust room temperatures based on weather forecasts and activity in the home, and
- intelligent home appliances.

According to statista.com, there are already over 23 billion IoT devices in use today, and there could be over 75 billion IoT devices in 2025.⁴⁶



I (*Fill-In*) The _____ was the precursor to today's Internet. **Answer:** ARPANET.

^{46. &}quot;Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025." Accessed November 1, 2020. https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

2 (*Fill-In*) The _____ (simply called "the web") is a collection of hardware and software associated with the Internet that allows computer users to locate and view documents (with various combinations of text, graphics, animations, audios and videos). **Answer:** World Wide Web.

3 (*Fill-In*) In the Internet of Things (IoT), a *thing* is any object with a(n) and the ability to send, and in some cases receive, data over the Internet. Answer: IP address.

1.12 Software Technologies

As you learn about and work in software development, you'll frequently encounter the following buzzwords:

- Refactoring: Reworking programs to make them clearer and easier to maintain while preserving their correctness and functionality. Many IDEs contain built-in refactoring tools to do major portions of the reworking automatically.
- Design patterns: Proven architectures for constructing flexible and maintainable object-oriented software. The field of design patterns tries to enumerate those recurring patterns, encouraging software designers to *reuse* them to develop better-quality software using less time, money and effort.
- Software Development Kits (SDKs)—The tools and documentation that developers use to program applications.

Self Check

I (*Fill-In*) ______ is the process of reworking programs to make them clearer and easier to maintain while preserving their correctness and functionality. **Answer:** refactoring.

1.13 How Big Is Big Data?

For computer scientists and data scientists, data is now as crucial as writing programs. According to IBM, approximately 2.5 quintillion bytes (2.5 exabytes) of data are created daily, 47 and 90% of the world's data was created in the last two years. 48 The Internet, which will play an important part in your career, is responsible for much of this trend. According to IDC, the global data supply will reach 175 zettabytes (equal to 175 trillion gigabytes or 175 billion terabytes) annually by 2025. 49 Consider the following examples of various popular data measures.

^{47. &}quot;Welcome to the world of A.I.." Accessed November 1, 2020. https://www.ibm.com/blogs/watson/2016/06/welcome-to-the-world-of-a-i/.

^{48. &}quot;Accelerate Research and Discovery." Accessed November 1, 2020. https://www.ibm.com/watson/advantages/accelerate.

^{49. &}quot;IDC: Expect 175 zettabytes of data worldwide by 2025." Accessed November 1, 2020. https://www.networkworld.com/article/3325397/storage/idc-expect-175-zetta-bytes-of-data-worldwide-by-2025.html.

Megabytes (MB)

One megabyte is about one million (actually 2^{20}) bytes. Many of the files we use daily require one or more MBs of storage. Some examples include:

- MP3 audio files—High-quality MP3s range from 1 to 2.4 MB per minute.⁵⁰
- Photos—JPEG format photos taken on a digital camera can require about 8 to 10 MB per photo.
- Video—Smartphone cameras can record video at various resolutions. Each minute of video can require many megabytes of storage. For example, on one of our iPhones, the Camera settings app reports that 1080p video at 30 frames-per-second (FPS) requires 130 MB/minute and 4K video at 30 FPS requires 350 MB/minute.

Gigabytes (GB)

One gigabyte is about 1000 megabytes (actually 2³⁰ bytes). A dual-layer DVD can store up to 8.5 GB⁵¹, which translates to:

- as much as 141 hours of MP3 audio,
- approximately 1000 photos from a 16-megapixel camera,
- approximately 7.7 minutes of 1080p video at 30 FPS, or
- approximately 2.85 minutes of 4K video at 30 FPS.

The current highest-capacity Ultra HD Blu-ray discs can store up to 100 GB of video.⁵² Streaming a 4K movie can use between 7 and 10 GB per hour (highly compressed).

Terabytes (TB)

One terabyte is about 1000 gigabytes (actually 2⁴⁰ bytes). Recent disk drives for desktop computers come in sizes up to 20 TB,⁵³ which is equivalent to:

- approximately 28 years of MP3 audio,
- approximately 1.68 million photos from a 16-megapixel camera,
- approximately 226 hours of 1080p video at 30 FPS, or
- approximately 84 hours of 4K video at 30 FPS.

Nimbus Data now has the largest solid-state drive (SSD) at 100 TB, which can store five times the 20-TB examples of audio, photos and video listed above.⁵⁴

^{50. &}quot;Audio File Size Calculations." Accessed November 1, 2020. https://www.audiomountain.com/tech/audio-file-size.html.

^{51. &}quot;DVD." Accessed November 1, 2020. https://en.wikipedia.org/wiki/DVD.

^{52. &}quot;Ultra HD Blu-ray." Accessed November 1, 2020. https://en.wikipedia.org/wiki/Ul-tra_HD_Blu-ray.

^{53. &}quot;History of hard disk drives." Accessed November 1, 2020. https://en.wikipedia.org/wiki/History_of_hard_disk_drives.

^{54. &}quot;Nimbus Data 100TB SSD — World's Largest SSD." Accessed November 1, 2020. https://www.cinema5d.com/nimbus-data-100tb-ssd-worlds-largest-ssd/.

Petabytes, Exabytes and Zettabytes

There are over four billion people online, creating about 2.5 quintillion bytes of data each day⁵⁵—that's 2500 petabytes (each petabyte is about 1000 terabytes) or 2.5 exabytes (each exabyte is about 1000 petabytes). A March 2016 *AnalyticsWeek* article stated that by 2021 there would be over 50 billion devices connected to the Internet (most of them through the Internet of Things; Section 1.11.4) and, by 2020, there would be 1.7 megabytes of new data produced per second for *every person on the planet*. ⁵⁶ At today's numbers (approximately 7.7 billion people⁵⁷), that's about

- 13 petabytes of new data per second,
- 780 petabytes per minute,
- 46,800 petabytes (46.8 exabytes) per hour, or
- 1,123 exabytes per day—that's 1.123 zettabytes (ZB) per day (each zettabyte is about 1000 exabytes).

That's the equivalent of over 5.5 million hours (over 600 years) of 4K video every day or approximately 116 billion photos every day!

Additional Big-Data Stats

For a real-time sense of big data, check out https://www.internetlivestats.com, with various statistics, including the numbers so far today of

- Google searches.
- Tweets.
- Videos viewed on YouTube.
- Photos uploaded on Instagram.

You can click each statistic to drill down for more information.

Some other interesting big-data facts:

- Every hour, YouTube users upload 30,000 hours of video, and almost 1 billion hours of video are watched on YouTube every day.⁵⁸
- Every second, there are 103,777 GBs (or 103.777 TBs) of Internet traffic, 9204 tweets sent, 87,015 Google searches and 86,617 YouTube videos viewed.⁵⁹

^{55. &}quot;How Much Data Is Created Every Day in 2020?" Accessed November 1, 2020. https://techjury.net/blog/how-much-data-is-created-every-day/#gref.

^{56. &}quot;Big Data Facts." Accessed November 1, 2020. https://analyticsweek.com/content/big-data-facts/.

^{57. &}quot;World Population." Accessed November 1, 2020. https://en.wikipedia.org/wiki/World_population.

^{58. &}quot;57 Fascinating and Incredible YouTube Statistics." Accessed November 1, 2020. https://www.brandwatch.com/blog/youtube-stats/.

^{59. &}quot;Tweets Sent in 1 Second." Accessed November 1, 2020. http://www.internetlivestats.com/one-second.

• On Facebook each day, there are 3.2 billion "likes" and comments, 60 and 5 billion emojis sent via Facebook Messenger. 61

Domo, Inc.'s infographic called "Data Never Sleeps 8.0" shows interesting statistics regarding how much data is generated *every minute*, including:⁶²

- 347,222 Instagram posts.
- 500 hours of video uploaded to YouTube.
- 147,000 photos uploaded to Facebook.
- 41,666,667 WhatsApp messages shared.
- 404,444 hours of Netflix video viewed.
- 479,452 users interact with Reddit content.
- 208,333 users participate in Zoom meetings.
- 1,388,889 people make video calls.

Computing Power Over the Years

Data is getting more massive, and so is the computing power for processing it. Today's processor performance is often measured in terms of FLOPS (floating-point operations per second). In the early to mid-1990s, the fastest supercomputer speeds were measured in gigaflops (10⁹ FLOPS). By the late 1990s, Intel produced the first teraflop (10¹² FLOPS) supercomputers. In the early-to-mid 2000s, speeds reached hundreds of teraflops, then in 2008, IBM released the first petaflop (10¹⁵ FLOPS) supercomputer. Currently, the fastest supercomputer—Fujitsu's Fugaku⁶³—is capable of 442 petaflops.⁶⁴

Distributed computing can link thousands of personal computers via the Internet to produce even more FLOPS. In late 2016, the Folding@home network—a distributed network in which people volunteer their personal computers' resources for use in disease research and drug design⁶⁵—was capable of over 100 petaflops. Companies like IBM are now working toward supercomputers capable of exaflops (10¹⁸ FLOPS).⁶⁶

^{60. &}quot;Facebook: 3.2 Billion Likes & Comments Every Day." Accessed November 1, 2020. https://marketingland.com/facebook-3-2-billion-likes-comments-every-day-19978.

^{61. &}quot;Facebook celebrates World Emoji Day by releasing some pretty impressive facts." Accessed November 1, 2020. https://mashable.com/2017/07/17/facebook-world-emoji-day/.

^{62. &}quot;Data Never Sleeps 8.0." Accessed November 1, 2020. https://www.domo.com/learn/data-never-sleeps-8.

^{63. &}quot;Top 500." Accessed December 24, 2020. https://en.wikipedia.org/wiki/TOP500#TOP_500.

^{64. &}quot;FLOPS." Accessed November 1, 2020. https://en.wikipedia.org/wiki/FLOPS.

^{65. &}quot;Folding@home." Accessed November 1, 2020. https://en.wikipedia.org/wiki/Folding@home.

^{66. &}quot;A new supercomputing-powered weather model may ready us for Exascale." Accessed November 1, 2020. https://www.ibm.com/blogs/research/2017/06/supercomputing-weather-model-exascale/.

The quantum computers now under development theoretically could operate at 18,000,000,000,000,000,000,000 times the speed of today's "conventional computers"! This number is so extraordinary that in one second, a quantum computer theoretically could do staggeringly more calculations than the total that have been done by all computers since the world's first computer appeared. This almost unimaginable computing power could wreak havoc with blockchain-based cryptocurrencies like Bitcoin. Engineers are already rethinking blockchain to prepare for such massive increases in computing power.⁶⁹

The history of supercomputing power is that it eventually works its way down from research labs—where extraordinary amounts of money have been spent to achieve those performance numbers—into "reasonably priced" commercial computer systems and even desktop computers, laptops, tablets and smartphones.

Computing power's cost continues to decline, especially with cloud computing. People used to ask the question, "How much computing power do I need on my system to deal with my *peak* processing needs?" Today, that thinking has shifted to "Can I quickly carve out on the cloud what I need *temporarily* for my most demanding computing chores?" You pay for only what you use to accomplish a given task.

Processing the World's Data Requires Lots of Electricity

Data from the world's Internet-connected devices is exploding, and processing that data requires tremendous amounts of energy. According to a recent article, energy use for processing data in 2015 was growing at 20% per year and consuming approximately three to five percent of the world's power. The article says that total data-processing power consumption could reach 20% by 2025.⁷⁰

Another enormous electricity consumer is the blockchain-based cryptocurrency Bitcoin. Processing just one Bitcoin transaction uses approximately the same amount of energy as powering the average American home for a week. The energy use comes from the process Bitcoin "miners" use to prove that transaction data is valid.⁷¹

According to some estimates, a year of Bitcoin transactions consumes more energy than many countries.⁷² Together, Bitcoin and Ethereum (another popular

^{67. &}quot;Only God can count that fast — the world of quantum computing." Accessed November 1, 2020. https://medium.com/@n.biedrzycki/only-god-can-count-that-fast-the-world-of-quantum-computing-406a0a91fcf4.

^{68. &}quot;Blockchain." Accessed December 24, 2020. https://en.wikipedia.org/wiki/Blockchain.

^{69. &}quot;Is Quantum Computing an Existential Threat to Blockchain Technology?" Accessed November 1, 2020. https://singularityhub.com/2017/11/05/is-quantum-computing-an-existential-threat-to-blockchain-technology/.

^{70. &}quot;Tsunami of data' could consume one fifth of global electricity by 2025." Accessed November 1, 2020. https://www.theguardian.com/environment/2017/dec/11/tsunami-of-data-could-consume-fifth-global-electricity-by-2025.

^{71. &}quot;One Bitcoin Transaction Consumes As Much Energy As Your House Uses in a Week." Accessed November 1, 2020. https://motherboard.vice.com/en_us/article/ywbbpm/bit-coin-mining-electricity-consumption-ethereum-energy-climate-change.

^{72. &}quot;Bitcoin Energy Consumption Index." Accessed November 1, 2020. https://digiconomist.net/bitcoin-energy-consumption.

blockchain-based platform and cryptocurrency) consume more energy per year than Finland, Belgium or Pakistan.⁷³

Morgan Stanley predicted in 2018 that "the electricity consumption required to create cryptocurrencies this year could actually outpace the firm's projected global electric vehicle demand—in 2025."⁷⁴ This situation is unsustainable, especially given the huge interest in blockchain-based applications, even beyond the cryptocurrency explosion. The blockchain community is working on fixes. 75,76

Big-Data Opportunities

The big-data explosion is likely to continue exponentially for years to come. With 50 billion computing devices on the horizon, we can only imagine how many more there will be over the next few decades. It's crucial for businesses, governments, the military, and even individuals to get a handle on all this data.

It's interesting that some of the best writings about big data, data science, artificial intelligence and more are coming out of prominent business organizations, such as J.P. Morgan, McKinsey, Bloomberg and the like. Big data's appeal to big business is undeniable, given the rapidly accelerating accomplishments. Many companies are making significant investments and getting valuable results through technologies like big data, machine learning and natural-language processing. This is forcing competitors to invest as well, rapidly increasing the need for computing professionals with computer-science and data-science experience. This growth is likely to continue for many years.



Self Check

- [(Fill-In) Today's processor performance is often measured in terms of _____. **Answer:** FLOPS (floating-point operations per second).
- (Fill-In) The technology that could wreak havoc with blockchain-based cryptocurrencies, like Bitcoin, and other blockchain-based technologies is _ **Answer:** quantum computers.
- **3** (True/False) With cloud computing you pay a fixed price for cloud services regardless of how much you use those services.

Answer: False. A key cloud-computing benefit is that you pay for only what you use to accomplish a given task.

^{73. &}quot;Ethereum Energy Consumption Index." Accessed November 1, 2020. https://digiconomist.net/ethereum-energy-consumption.

^{74. &}quot;Power Play: What Impact Will Cryptocurrencies Have on Global Utilities?" Accessed November 1, 2020. https://www.morganstanley.com/ideas/cryptocurrencies-global-utili-

^{75. &}quot;Blockchains Use Massive Amounts of Energy—But There's a Plan to Fix That." Accessed November 1, 2020. https://www.technologyreview.com/s/609480/bitcoin-uses-massiveamounts-of-energybut-theres-a-plan-to-fix-it/.

^{76. &}quot;How to fix Bitcoin's energy-consumption problem." Accessed November 1, 2020. http:// mashable.com/2017/12/01/bitcoin-energy/.

1.13.1 Big-Data Analytics

Data analytics is a mature and well-developed discipline. The term "data analysis" was coined in 1962,⁷⁷ though people have been analyzing data using statistics for thousands of years, going back to the ancient Egyptians.⁷⁸ Big-data analytics is a more recent phenomenon—the term "big data" was coined around 1987.⁷⁹

Consider four of the V's of big data^{80,81}:

- 1. Volume—the data the world is producing is growing exponentially.
- 2. Velocity—the speed at which data is being produced, the speed at which it moves through organizations and the speed at which data changes are growing quickly. 82,83,84
- 3. Variety—data used to be alphanumeric (that is, consisting of alphabetic characters, digits, punctuation and some special characters)—today, it also includes images, audios, videos and data from an exploding number of Internet of Things sensors in our homes, businesses, vehicles, cities and more.
- **4.** Veracity—the validity of the data—is it complete and accurate? Can we trust that data when making crucial decisions? Is it real?

Most data is now being created digitally in a *variety* of types, in extraordinary *volumes* and moving at astonishing *velocities*. Moore's Law and related observations have enabled us to store data economically and process and move it faster—and all at rates growing exponentially over time. Digital data storage has become so vast in capacity, and so cheap and small, that we can now conveniently and economically retain *all* the digital data we're creating.⁸⁵ That's big data.

^{77. &}quot;A Very Short History Of Data Science." Accessed November 1, 2020. https://www.forbes.com/sites/gilpress/2013/05/28/a-very-short-history-of-data-science/.

^{78. &}quot;A Brief History of Data Analysis." Accessed November 1, 2020. https://www.flydata.com/blog/a-brief-history-of-data-analysis/.

^{79.} Diebold, Francis. (2012). On the Origin(s) and Development of the Term "Big Data". SSRN Electronic Journal. 10.2139/ssrn.2152421. https://www.researchgate.net/publication/255967292_On_the_Origins_and_Development_of_the_Term_'Big_Data'.

^{80. &}quot;The Four V's of Big Data." Accessed November 1, 2020. https://www.ibmbigdatahub.com/infographic/four-vs-big-data.

^{81.} There are lots of articles and papers that add many other "V-words" to this list.

^{82. &}quot;Volume, velocity, and variety: Understanding the three V's of big data." Accessed November 1, 2020. https://www.zdnet.com/article/volume-velocity-and-variety-understand-ing-the-three-vs-of-big-data/.

^{83. &}quot;3Vs (volume, variety and velocity)." Accessed November 1, 2020. https://whatis.techtar-get.com/definition/3Vs.

^{84. &}quot;Big Data: Forget Volume and Variety, Focus On Velocity." Accessed November 1, 2020. https://www.forbes.com/sites/brentdykes/2017/06/28/big-data-forget-volume-and-variety-focus-on-velocity.

^{85. &}quot;How Much Information Is There In the World?" Accessed November 1, 2020. http://www.lesk.com/mlesk/ksg97/ksg.html. [The following article pointed us to this Michael Lesk article: https://www.forbes.com/sites/gilpress/2013/05/28/a-very-short-history-of-data-science/.]

46 Chapter I Introduction to Computers and C

The following Richard W. Hamming quote—although from 1962—sets the tone for the rest of this book:

"The purpose of computing is insight, not numbers."86

Data science is producing new, deeper, subtler and more valuable insights at a remarkable pace. It's truly making a difference. Big-data analytics is an integral part of the answer.

To get a sense of big data's scope in industry, government and academia, check out the high-resolution graphic⁸⁷—you can click to zoom for easier readability:

http://mattturck.com/wp-content/uploads/2018/07/ Matt_Turck_FirstMark_Big_Data_Landscape_2018_Final.png

1.13.2 Data Science and Big Data Are Making a Difference: Use Cases

The data-science field is growing rapidly because it's producing significant results that are making a difference. We enumerate data-science and big-data use cases in the following table. We expect that the use cases and our examples, exercises and projects will inspire interesting term projects, directed-study projects, capstone-course projects and thesis research. Big-data analytics has resulted in improved profits, better customer relations, and even sports teams winning more games and championships while spending less on players. ^{88,89,90}

Data-science use cases

anomaly detection assisting people with disabilities automated closed captioning brain mapping cancer diagnosis/treatment classifying handwriting computer vision credit scoring
crime prevention
CRISPR gene editing
crop-yield improvement
customer churn and retention
customer service agents
cybersecurity

data mining
data visualization
diagnostic medicine
dynamic driving routes
dynamic pricing
electronic health records
emotion detection

^{86.} Hamming, R. W., *Numerical Methods for Scientists and Engineers* (New York: McGraw Hill, 1962). [The following article pointed us to Hamming's book and his quote that we cited: https://www.forbes.com/sites/gilpress/2013/05/28/a-very-short-history-of-data-science/.]

^{87.} Turck, M., and J. Hao, "Great Power, Great Responsibility: The 2018 Big Data & AI Landscape," http://mattturck.com/bigdata2018/.

^{88.} Sawchik, T., Big Data Baseball: Math, Miracles, and the End of a 20-Year Losing Streak (New York: Flat Iron Books, 2015).

^{89.} Ayres, I., Super Crunchers (Bantam Books, 2007), pp. 7–10.

^{90.} Lewis, M., Moneyball: The Art of Winning an Unfair Game (W. W. Norton & Company, 2004).

Data-science use cases

facial recognition fraud detection game playing health outcome improvement human genome sequencing identity-theft prevention immunotherapy intelligent assistants Internet of Things (IoT) and medical device monitoring inventory control language translation location-based services malware detection marketing analytics natural-language translation new pharmaceuticals personal assistants

personalized medicine phishing elimination pollution reduction precision medicine predicting disease outbreaks predicting health outcomes predicting weather-sensitive product sales preventative medicine preventing disease outbreaks real-estate valuation recommendation systems ride-sharing risk minimization robo financial advisors saving energy self-driving cars sentiment analysis

sharing economy similarity detection smart cities smart homes smart meters smart thermostats smart traffic control social graph analysis spam detection stock market forecasting summarizing text telemedicine terrorist attack prevention theft prevention trend spotting visual product search voice recognition weather forecasting

1.14 Case Study—A Big-Data Mobile Application

In your career, you'll work with many programming languages and software technologies. With its 130 million monthly active users, 91 Google's Waze GPS navigation app is one of the most widely used big-data apps. Early GPS navigation devices and apps relied on static maps and GPS coordinates to determine the best route to your destination. They could not adjust dynamically to changing traffic situations.

Waze processes massive amounts of crowdsourced data—that is, the data that's continuously supplied by their users and their users' devices worldwide. They analyze this data as it arrives to determine the best route to get you safely to your destination in the least amount of time. To accomplish this, Waze relies on your smartphone's Internet connection. The app automatically sends location updates to their servers (assuming you allow it to). They use that data to dynamically re-route you based on current traffic conditions and to tune their maps. Users report other information, such as roadblocks, construction, obstacles, vehicles in breakdown lanes, police locations, gas prices and more. Waze then alerts other drivers in those locations.

Waze uses many technologies to provide its services. We're not privy to how Waze is implemented, but we infer below a list of technologies they probably use. For example,

 Most apps created today use at least some open-source software. You'll take advantage of open-source libraries and tools in the case studies.

^{91. &}quot;Waze Communities." Accessed November 1, 2020. https://www.waze.com/communities.

- Waze communicates information over the Internet between their servers and their users' mobile devices. Today, such data typically is transmitted in JSON (JavaScript Object Notation) format. Often the JSON data will be hidden from you by the libraries you use.
- Waze uses speech synthesis to speak driving directions and alerts to you, and uses speech recognition to understand your spoken commands. Many cloud vendors provide speech-synthesis and speech-recognition capabilities.
- Once Waze converts a spoken natural-language command to text, it determines the action to perform, which requires natural language processing (NLP).
- Waze displays dynamically updated visualizations, such as alerts and interactive maps.
- Waze uses your phone as a streaming Internet of Things (IoT) device. Each phone is a GPS sensor that continuously streams data over the Internet to Waze.
- Waze receives IoT streams from millions of phones at once. It must process, store and analyze that data immediately to update your device's maps, display and speak relevant alerts and possibly update your driving directions. This requires massively parallel processing capabilities implemented with clusters of computers in the cloud. You can use various big-data infrastructure technologies to receive streaming data, store that big data in appropriate databases and process the data with software and hardware that provide massively parallel processing capabilities.
- Waze uses artificial-intelligence capabilities to perform the data-analysis tasks
 that enable it to predict the best routes based on the information it receives.
 You can use machine learning and deep learning, respectively, to analyze massive amounts of data and make predictions based on that data.
- Waze probably stores its routing information in a graph database. Such databases can efficiently calculate shortest routes. You can use graph databases, such as Neo4J.
- Many cars are equipped with devices that help them "see" cars and obstacles around them. These are used to help implement automated braking systems and are a key part of self-driving car technology. Rather than relying on users to report obstacles and stopped cars on the side of the road, navigation apps could take advantage of cameras and other sensors by using deep-learning computer-vision techniques to analyze images "on the fly" and automatically report those items. You can use deep learning for computer vision.

1.15 Al—at the Intersection of Computer Science and Data Science

When a baby first opens its eyes, does it "see" its parent's faces? Does it understand any notion of what a face is—or even what a simple shape is? Babies must "learn" the

world around them. That's what artificial intelligence (AI) is doing today. It's looking at massive amounts of data and learning from it. AI is being used to play games, implement a wide range of computer-vision applications, enable self-driving cars, enable robots to learn to perform new tasks, diagnose medical conditions, translate speech to other languages in near real-time, create chatbots that can respond to arbitrary questions using massive databases of knowledge, and much more. Who'd have guessed just a few years ago that artificially intelligent self-driving cars would be allowed on our roads—or even become common? Yet, this is now a highly competitive area. The ultimate goal of all this learning is artificial general intelligence—an AI that can perform intelligence tasks as well as humans can.

Artificial-Intelligence Milestones

Several artificial-intelligence milestones, in particular, captured people's attention and imagination, made the general public start thinking that AI is real and made businesses think about commercializing AI:

- In a 1997 match between IBM's DeepBlue computer system and chess Grandmaster Gary Kasparov, DeepBlue became the first computer to beat a reigning world chess champion under tournament conditions. ⁹² IBM loaded DeepBlue with hundreds of thousands of grandmaster chess games. DeepBlue was capable of using *brute force* to evaluate up to 200 million moves per second! ⁹³ This is big data at work. IBM received the Carnegie Mellon University Fredkin Prize, which in 1980 offered \$100,000 to the creators of the first computer to beat a world chess champion. ⁹⁴
- In 2011, IBM's Watson beat the two best human Jeopardy! players in a \$1 million match. Watson simultaneously used hundreds of language-analysis techniques to locate correct answers in 200 million pages of content (including all of Wikipedia) requiring four terabytes of storage. 95,96 Watson was trained with machine-learning and reinforcement-learning techniques. 97 Powerful libraries enable you to perform machine-learning and reinforcement-learning in various programming languages.

^{92. &}quot;Deep Blue versus Garry Kasparov." Accessed November 1, 2020. https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov.

^{93. &}quot;Deep Blue (chess computer)." Accessed November 1, 2020. https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer).

^{94. &}quot;IBM Deep Blue Team Gets \$100,000 Prize." Accessed November 1, 2020. https://arti-cles.latimes.com/1997/jul/30/news/mn-17696.

^{95. &}quot;IBM Watson: The inside story of how the Jeopardy-winning supercomputer was born, and what it wants to do next." Accessed November 1, 2020. https://www.techrepublic.com/article/ibm-watson-the-inside-story-of-how-the-jeopardy-winning-supercomputer-was-born-and-what-it-wants-to-do-next/.

^{96. &}quot;Watson (computer)." Accessed November 1, 2020. https://en.wikipedia.org/wiki/Watson_(computer).

^{97. &}quot;Building Watson: An Overview of the DeepQA Project." Accessed November 1, 2020. https://www.aaai.org/Magazine/Watson/watson.php, *AI Magazine*, Fall 2010.

- **50**
- Go—a board game created in China thousands of years ago⁹⁸—is widely considered one of the most complex games ever invented with 10¹⁷⁰ possible board configurations.⁹⁹ To give you a sense of how large a number that is, it's believed that there are (only) between 10⁷⁸ and 10⁸² atoms in the known universe!^{100,101} In 2015, AlphaGo—created by Google's DeepMind group—used *deep learning with two neural networks to beat the European Go champion Fan Hui*. Go is considered to be a far more complex game than chess. Powerful libraries enable you to use neural networks for deep learning.
- More recently, Google generalized its AlphaGo AI to create AlphaZero—a game-playing AI that *teaches itself to play other games*. In December 2017, AlphaZero learned the rules of and taught itself to play chess in less than four hours using reinforcement learning. It then beat the world champion chess program, Stockfish 8, in a 100-game match—winning or drawing every game. After *training itself* in Go for just eight hours, AlphaZero was able to play Go vs. its AlphaGo predecessor, winning 60 of 100 games. ¹⁰²

AI: A Field with Problems But No Solutions

For many decades, AI has been a field with problems and *no* solutions. That's because once a particular problem is solved, people say, "Well, that's not intelligence; it's just a computer program that tells the computer exactly what to do." However, with machine learning, deep learning and reinforcement learning, we're not pre-programming solutions to *specific* problems. Instead, we're letting our computers solve problems by learning from data—and, typically, lots of it. Many of the most interesting and challenging problems are being pursued with deep learning. Google alone has thousands of deep-learning projects underway. 103,104

Self Check

I (*Fill-In*) The ultimate goal of AI is to produce a(n) _____. Answer: artificial general intelligence.

- 98. "A Brief History of Go." Accessed November 1, 2020. http://www.usgo.org/brief-history-go.
- 99. "Google artificial intelligence beats champion at world's most complicated board game." Accessed November 1, 2020. https://www.pbs.org/newshour/science/google-artificial-intelligence-beats-champion-at-worlds-most-complicated-board-game.
- 100. "How Many Atoms Are There in the Universe?" Accessed November 1, 2020. https://www.universetoday.com/36302/atoms-in-the-universe/.
- 101. "Observable universe." Accessed November 1, 2020. https://en.wikipedia.org/wiki/0b-servable_universe#Matter_content.
- 102. "AlphaZero AI beats champion chess program after teaching itself in four hours." Accessed November 1, 2020. https://www.theguardian.com/technology/2017/dec/07/alphazero-google-deepmind-ai-beats-champion-program-teaching-itself-to-play-four-hours.
- 103. "Google has more than 1,000 artificial intelligence projects in the works." Accessed November 1, 2020. http://theweek.com/speedreads/654463/google-more-than-1000-artificial-intelligence-projects-works.
- 104. "Google says 'exponential' growth of AI is changing nature of compute." Accessed November 1, 2020. https://www.zdnet.com/article/google-says-exponential-growth-of-ai-is-changing-nature-of-compute/.

 2 (Fill-In) IBM's Watson beat the two best human Jeopardy! players. Watson watrained using a combination of learning and learning techniques Answer: machine, reinforcement. 3 (Fill-In) Google's taught itself to play chess in less than four hours usin reinforcement learning, then beat the world champion chess program, Stockfish 8, i a 100-game match—winning or drawing every game. Answer: AlphaZero.
Self-Review Exercises
 Fill in the blanks in each of the following statements: a) Computers process data under the control of instructions called b) A computer's key logical units are: unit, unit unit, unit, unit and unit. c) The three types of programming languages discussed in the chapter ar, and d) Programs that translate high-level-language programs into machine lar guage are called e) is an operating system for mobile devices based on the Linux kerner f) A(n) allows a device to respond to motion. g) C is widely known as the language of the operating system.
 Fill in the blanks in each of the following sentences about the C environmenta. C programs are normally typed into a computer using a(n) In a C system, a(n) automatically executes before the translation phase begins. The combines the output of the compiler with various librar functions to produce an executable image. The transfers the executable image from disk to memory.
Answers to Self-Review Exercises
a) programs. b) input, output, memory, central processing, arithmetic and logic, secondary storage. c) machine languages, assembly languages, high-level languages. d) compilers. e) Android. f) acceleromoter. g) UNIX.

- 1.2 a) editor. b) preprocessor. c) linker. d) loader.

Exercises

- 1.3 Categorize each of the following items as either hardware or software:
 - a) CPU.
 - b) C compiler.
 - c) ALU.
 - d) C preprocessor.
 - e) input unit.
 - f) an editor program.

- 1.5 Discuss the purpose of each of the following:
 - a) stdin
 - b) stdout
 - c) stderr
- 1.6 (Gender Neutrality) Write the steps of a manual procedure to process a text paragraph and replace gender-specific words with gender-neutral ones. Assuming you've been given a list of gender-specific words and their gender-neutral replacements (e.g., replace "wife" or "husband" with "spouse," replace "man" or "woman" with "person," replace "daughter" or "son" with "child," and so on), explain the procedure you'd use to read through a paragraph of text and manually perform these replacements. How might your procedure generate a strange term like "woperchild" and how might you modify your procedure to avoid this possibility? In Chapter 3, you'll learn that a more formal computing term for "procedure" is "algorithm," and that an algorithm specifies the steps to be performed and the order in which to perform them.
- **1.7** (Self-Driving Cars) Just a few years back, the notion of driverless cars on our streets would have seemed impossible (in fact, our spell-checking software doesn't recognize the word "driverless"). Many of the technologies you'll study in this book are making self-driving cars possible. They're already common in some areas.
 - a) If you hailed a taxi and a driverless taxi stopped for you, would you get into the back seat? Would you feel comfortable telling it where you want to go and trusting that it would get you there? What safety measures would you want in place? What would you do if the car headed off in the wrong direction?
 - b) What if two self-driving cars approached a one-lane bridge from opposite directions? What protocol should they go through to determine which car should proceed?
 - c) What if you're behind a car stopped at a red light, the light turns green, and the car doesn't move? You honk, and nothing happens. You get out of your car and notice that there's no driver. What would you do?
 - d) If a police officer pulls over a speeding self-driving car in which you're the only passenger, who—or what entity—should pay the ticket?

- e) One serious concern with self-driving vehicles is that they could potentially be hacked. Someone could set the speed high (or low), which could be dangerous. What if they redirect you to a destination other than what you want?
- **1.8** (Research: Reproducibility) A crucial concept in data-science studies is reproducibility, which helps others (and you) reproduce your results. Research reproducibility and list the concepts used to create reproducible results in data-science studies.
- 1.9 (Research: Artificial General Intelligence) One of the most ambitious goals in the field of AI is to achieve artificial general intelligence—the point at which machine intelligence would equal human intelligence. Research this intriguing topic. When is this forecast to happen? What are some key ethical issues this raises? Human intelligence seems to be stable over long periods. Powerful computers with artificial general intelligence could conceivably (and quickly) evolve intelligence far beyond that of humans. Research and discuss the issues this raises.
- **1.10** (Research: Intelligent Assistants) Many companies now offer computerized intelligent assistants, such as IBM Watson, Amazon Alexa, Apple Siri, Google Assistant and Microsoft Cortana. Research these and others and list uses that can improve people's lives. Research privacy and ethics issues for intelligent assistants. Locate amusing intelligent-assistant anecdotes.
- **1.11** (Research: AI in Health Care) Research the rapidly growing field of AI big-data applications in health care. For example, suppose a diagnostic medical application had access to every x-ray that's ever been taken and the associated diagnoses—that's surely big data. "Deep Learning" computer-vision applications can work with this "labeled" data to learn to diagnose medical problems. Research deep learning in diagnostic medicine and describe some of its most significant accomplishments. What are some ethical issues of having machines instead of human doctors performing medical diagnoses? Would you trust a machine-generated diagnosis? Would you ask for a second opinion?
- **1.12** (Research: Privacy and Data Integrity Legislation) In the Preface, we mentioned HIPAA (Health Insurance Portability and Accountability Act) and the California Consumer Privacy Act (CCPA) in the United States and GDPR (General Data Protection Regulation) for the European Union. Laws like these are becoming more common and stricter. Investigate each of these laws and their effects on your privacy.
- **1.13** (Research: Personally Identifiable Information) Protecting users personally identifiable information (PII) is an important aspect of privacy. Research and comment on this issue.
- **1.14** (Research: Big Data, AI and the Cloud—How Companies Use These Technologies) For a major organization of your choice, research how they may be using each of the following technologies: AI, big data, the cloud, mobile, natural-language processing, speech recognition, speech synthesis, database, machine learning, deep learning, reinforcement learning, Hadoop, Spark, Internet of Things (IoT) and web services.

- **1.15** (Research: Raspberry Pi and the Internet of Things) It's now possible to have a computer at the heart of just about any device and to connect those devices to the Internet. This has led to the Internet of Things (IoT), which interconnects billions of devices. The Raspberry Pi is an economical computer often at the heart of IoT devices. Research the Raspberry Pi and some of the many IoT applications in which it's used.
- **1.16** (Research: The Ethics of Deep Fakes) Artificial-intelligence technologies make it possible to create deep fakes—realistic fake videos of people that capture their appearance, voice, body motions and facial expressions. You can have them say and do whatever you specify. Research the ethics of deep fakes. What would happen if you turned on your TV and saw a deep-fake video of a prominent government official or newscaster reporting that a nuclear attack was about to happen? Research Orson Welles and his "War of the Worlds" radio broadcast of 1938, which created mass panic.
- **1.17** (Research: Blockchain—A World of Opportunity) Cryptocurrencies like Bitcoin and Ethereum are based on a technology called blockchain that has seen explosive growth over the last few years. Research blockchain's origin, applications and how it came to be used as the basis for cryptocurrencies. Research other major applications of blockchain. Over the next many years, there will be extraordinary opportunities for software developers who thoroughly understand blockchain applications development.
- 1.18 (Research: Secure C and the CERT Division of Carnegie Mellon University's Software Engineering Institute) Experience has shown that it's challenging to build industrial-strength systems that stand up to attacks. Such attacks can be instantaneous and global in scope. Many of the world's largest companies, government agencies, and military organizations have had their systems compromised. Such vulnerabilities often come from simple programming issues. Building security into software from the start of its development can significantly reduce vulnerabilities. Carnegie Mellon University's Software Engineering Institute (SEI) created CERT (https://www.sei.cmu.edu/about/divisions/cert/index.cfm) to analyze and respond promptly to attacks. CERT publishes and promotes secure coding standards to help C programmers and others implement industrial-strength systems that avoid the programming practices that leave systems vulnerable to attacks. The CERT standards evolve as new security issues arise. The SEI CERT C Coding Standard is concerned with "hardening" computer systems and applications to resist attacks. Research CERT and discuss their accomplishments and current challenges. To help you focus on secure C coding practices, Chapters 2–12 and 14 contain Secure C Coding sections that present some key issues and techniques and provide links and references so that you can continue learning.
- **1.19** (*Research: IBM Watson*) IBM is partnering with tens of thousands of companies—including our publisher, Pearson Education—across a wide range of industries. Research some of IBM Watson's key accomplishments and the kinds of challenges IBM and its partners are addressing.

Intro to C Programming



2

Objectives

In this chapter, you'll:

- Write simple C programs.
- Use simple input and output statements.
- Use the fundamental data types.
- Learn computer memory concepts.
- Use arithmetic operators.
- Learn the precedence of arithmetic operators.
- Write simple decision-making statements.
- Begin focusing on secure C programming practices.

- 2.1 Introduction
- **2.2** A Simple C Program: Printing a Line of Text
- **2.3** Another Simple C Program: Adding Two Integers
- **2.4** Memory Concepts
- **2.5** Arithmetic in C
- **2.6** Decision Making: Equality and Relational Operators
- **2.7** Secure C Programming

Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises

2.1 Introduction

The C language facilitates a structured and disciplined approach to computer-program design. This chapter introduces C programming and presents several examples illustrating many fundamental C features. We analyze each example one statement at a time. In Chapters 3 and 4, we introduce structured programming—a methodology that will help you produce clear, easy-to-maintain programs. We then use the structured approach throughout the remainder of the text. This chapter concludes with the first of our "Secure C Programming" sections.

2.2 A Simple C Program: Printing a Line of Text

We begin with a simple C program that prints a line of text. The program and its screen output are shown in Fig. 2.1.

```
// fig02_01.c
// A first program in C.
#include <stdio.h>
// function main begins program execution
int main(void) {
   printf("Welcome to C!\n");
} // end function main
Welcome to C!
```

Fig. 2.1 A first program in C.

Comments

```
Lines 1 and 2
    // fig02_01.c
    // A first program in C.
```

begin with //, indicating that these two lines are comments. You insert comments to document programs and improve program readability. Comments do not cause the computer to perform actions when you execute programs—they're simply ignored. It's our convention in each program to use the line 1 comment to specify the file-

name, and the line 2 comment to describe the program's purpose. Comments also help other people read and understand your program.

You can also use /*...*/ multi-line comments in which everything from /* on the first line to */ at the end of the last line is a comment. We prefer the shorter // comments because they eliminate common programming errors that occur with / \bigotimes ERR *...*/ comments, such as accidentally omitting the closing */.

#include Preprocessor Directive

Line 3

#include <stdio.h>

is a C preprocessor directive. The preprocessor handles lines beginning with # before compilation. Line 3 tells the preprocessor to include the contents of the standard input/output header (<stdio.h>). This is a file containing information the compiler uses to ensure that you correctly use standard input/output library functions such as printf (line 7). Chapter 5 explains the contents of headers in more detail.

Blank Lines and White Space

We simply left line 4 blank. You use blank lines, space characters and tab characters to make programs easier to read. Together, these are known as white space and are generally ignored by the compiler.

The main Function

Line 6

```
int main(void) {
```

is a part of every C program. The parentheses after main indicate that main is a program building block called a function. C programs consist of functions, one of which must be main. Every program begins executing at the function main. As a good practice, precede every function by a comment (as in line 5) stating the function's purpose.

Functions can return information. The keyword int to the left of main indicates that main "returns" an integer (whole number) value. We'll explain what it means for a function to "return a value" in Chapter 4 when we use a math function to perform a calculation and in Chapter 5 when we create custom functions. For now, simply include the keyword int to the left of main in each of your programs.

Functions also can receive information when they're called upon to execute. The void in parentheses here means that main does not receive any information. In Chapter 15, we'll show an example of main receiving information.

A left brace, {, begins each function's body (end of line 6). A corresponding right brace, }, ends each function's body (line 8). When a program reaches main's closing right brace, the program terminates. The braces and the portion of the program between them form a *block*—an important program unit that we'll discuss more in subsequent chapters.

An Output Statement

Line 7

```
printf("Welcome to C!\n");
```

instructs the computer to perform an action, namely to display on the screen the string of characters enclosed in the quotation marks. A string is sometimes called a character string, a message or a literal.

The entire line 7—including the "call" to the printf function to perform its task, the printf's argument within the parentheses and the semicolon (;)—is called a statement. Every statement must end with a semicolon statement terminator. The "f" in printf stands for "formatted." When line 7 executes, it displays the message Welcome to C! on the screen. The characters usually print as they appear between the double quotes, but notice that the characters \n were not displayed.

Escape Sequences

In a string, the backslash (\) is an escape character. It indicates that printf should do something out of the ordinary. In a string, the compiler combines a backslash with the next character to form an escape sequence. The escape sequence \n means newline. When printf encounters a newline in a string, it positions the output cursor to the beginning of the next line. Some common escape sequences are listed below:

Escape sequence	Description
\n	Moves the cursor to the beginning of the next line.
\t	Moves the cursor to the next horizontal tab stop.
\a	Produces a sound or visible alert without changing the current cursor position.
\\	Because the backslash has special meaning in a string, \\ is required to insert a backslash character in a string.
\"	Because strings are enclosed in double quotes, \" is required to insert a double-quote character in a string.

The Linker and Executables

Standard library functions like printf and scanf are not part of the C programming language. For example, the compiler cannot find a spelling error in printf or scanf. When compiling a printf statement, the compiler merely provides space in the object program for a "call" to the library function. But the compiler does not know where the library functions are—the linker does. When the linker runs, it locates the library functions and inserts the proper calls to these functions in the object program. Now the object program is complete and ready to execute. The linked program is called an executable. If the function name is misspelled, the linker will spot the error—it will not be able to match the name in the program with the name of any known function in the libraries.



Indentation Conventions

Indent the entire body of each function one level of indentation (we recommend three spaces) within the braces that define the function's body. This indentation emphasizes a program's functional structure and helps make them easier to read.

Set a convention for the indent size you prefer and uniformly apply that convention. The tab key may be used to create indents, but tab stops can vary. Professional style guides often recommend using spaces rather than tabs. Some code editors actually insert spaces when you press the *Tab* key.

Using Multiple printfs

The printf function can display Welcome to C! several different ways. For example, Fig. 2.2 uses two statements to produce the same output as Fig. 2.1. This works because each printf resumes printing where the previous one finished. Line 7 displays Welcome followed by a space (but no newline). Line 8's printf begins printing on the same line immediately following the space.

```
// fig02_02.c
// Printing on one line with two printf statements.
#include <stdio.h>

// function main begins program execution
int main(void) {
   printf("Welcome ");
   printf("to C!\n");
} // end function main
Welcome to C!
```

Fig. 2.2 Printing one line with two printf statements.

Displaying Multiple Lines with a Single printf

One printf can display several lines, as in Fig. 2.3. Each \n moves the output cursor to the beginning of the next line.

```
1  // fig02_03.c
2  // Printing multiple lines with a single printf.
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main(void) {
7    printf("Welcome\nto\nC!\n");
8 } // end function main
Welcome
```

Fig. 2.3 | Printing multiple lines with a single printf.

to C!



I (Multiple Choice) Consider the code:

```
int main(void)
```

Which of the following statements is *false*?

- a) The parentheses after main indicate that it is a function.
- b) The keyword int to the left of main indicates that main returns an integer value, and the void in parentheses means that main does not receive any information.
- c) A left parenthesis, (, begins every function's body. A corresponding right parenthesis,), ends each function's body.
- d) When execution reaches the end of main, the program terminates.

Answer: c) is *false*. Actually, a left brace, {, begins every function's body, and a corresponding right brace, }, ends each function's body.

- **2** (*Multiple Choice*) Which of the following statements is *false*?
 - a) Each printf resumes printing where the previous one stopped printing.
 - b) In the following code, the first printf displays Welcome followed by a space, and the second printf begins printing on the next line of output:

```
printf("Welcome ");
printf("to C!\n");
```

c) The following printf prints several lines of text:

```
printf("Welcome\nto\nC!\n");
```

d) Each time a \n escape sequence is encountered, output continues at the beginning of the next line.

Answer: b) is *false*. Actually, the second printf begins printing immediately following the space output by the first printf.

2.3 Another Simple C Program: Adding Two Integers

Our next program uses the scanf standard library function to obtain two integers typed by a user at the keyboard, then computes their sum and displays the result using printf. The program and sample output are shown in Fig. 2.4. In the input/output dialog box of Fig. 2.4, we emphasize the numbers entered by the user in **bold**.

```
// fig02_04.c
// Addition program.
#include <stdio.h>

// function main begins program execution
int main(void) {
  int integer1 = 0; // will hold first number user enters
  int integer2 = 0; // will hold second number user enters
```

Fig. 2.4 | Addition program. (Part 1 of 2.)

```
9
       printf("Enter first integer: "); // prompt
10
       scanf("%d", &integer1); // read an integer
11
12
       printf("Enter second integer: "); // prompt
13
       scanf("%d", &integer2); // read an integer
14
15
       int sum = 0; // variable in which sum will be stored
16
       sum = integer1 + integer2; // assign total to sum
17
18
       printf("Sum is %d\n", sum); // print sum
19
    } // end function main
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 2.4 | Addition program. (Part 2 of 2.)

The comment in line 2 states the program's purpose. Again, the program begins execution with main (lines 6–20)—the braces at lines 6 and 20 mark the beginning and end of main's body, respectively.

Variables and Variable Definitions

Lines 7 and 8

```
int integer1 = 0; // will hold first number user enters
int integer2 = 0; // will hold second number user enters
```

are definitions. The names integer1 and integer2 are variables—locations in memory where the program can store values for later use. These definitions specify that integer1 and integer2 have type int. This means they'll hold whole-number integer values, such as 7, -11, 0 and 31914. Lines 7 and 8 initialize each variable to 0 by following the variable's name with an = and a value. Although it's not necessary to explicitly initialize every variable, doing so will help avoid many common problems.

Define Variables Before They Are Used

All variables must be defined with a name and a type before they can be used in a program. You can place each variable definition anywhere in main before that variable's first use in the code. In general, you should define variables close to their first use.

Identifiers and Case Sensitivity

A variable name can be any valid identifier. Each identifier may consist of letters, digits and underscores (_), but may not begin with a digit. C is case sensitive, so a1 and A1 are different identifiers. A variable name should start with a lowercase letter. Later in the text, we'll assign special significance to identifiers that begin with a capital letter and identifiers that use all capital letters.

Choosing meaningful variable names helps make a program self-documenting, so fewer comments are needed. Avoid starting identifiers with an underscore (_) to pre-

vent conflicts with compiler-generated identifiers and standard library identifiers. Multiple-word variable names can make programs more readable. For such names:

- separate the words with underscores, as in total_commissions, or
- run the words together and begin each subsequent word with a capital letter, as in totalCommissions.

The latter style is called camel casing because the pattern of uppercase and lowercase letters resembles a camel's silhouette. We prefer camel casing.

Prompting Messages

Line 10

```
printf("Enter first integer: "); // prompt
```

displays "Enter first integer: ". This message is called a prompt because it tells the user to take a specific action.

The scanf Function and Formatted Inputs

Line 11

```
scanf("%d", &integer1); // read an integer
```

uses scanf to obtain a value from the user. The function reads from the standard input, which is usually the keyboard.

The "f" in scanf stands for "formatted." This scanf has two arguments—"%d" and &integer1. The "%d" is the format control string. It indicates the type of data the user should enter. The %d conversion specification specifies that the data should be an integer—the d stands for "decimal integer". A % character begins each conversion specification.

scanf's second argument begins with an ampersand (&) followed by the variable name. The & is the address operator and, when combined with the variable name, tells scanf the location (or address) in memory of the variable integer1. scanf then stores the value the user enters at that memory location.

Using the ampersand (&) is often confusing to novice programmers and people who have programmed in other languages that do not require this notation. For now, just remember to precede each variable in every call to scanf with an ampersand. Some exceptions to this rule are discussed in Chapters 6 and 7. The use of & will become clear after we study pointers in Chapter 7.

Forgetting the ampersand (&) before a variable in a scanf statement typically ERR (X) results in an execution-time error. On many systems, this causes a "segmentation fault" or "access violation." Such an error occurs when a user's program attempts to access a part of the computer's memory to which it does not have access privileges. The precise cause of this error will be explained in Chapter 7.

> When line 11 executes, the computer waits for the user to enter a value for integer1. The user types an integer, then presses the *Enter* key (or *Return* key) to send the number to the computer. The computer then places the number (or value) in integer1. Any subsequent references to integer1 in the program use this same value.

Functions printf and scanf facilitate interaction between the user and the computer. This interaction resembles a dialogue and is often called **interactive computing**.

Prompting for and Inputting the Second Integer

Line 13

Defining the sum Variable

Line 16

```
int sum = 0; // variable in which sum will be stored
```

defines the int variable sum and initializes it to 0 before we use sum in line 17.

Assignment Statement

The assignment statement in line 17

```
sum = integer1 + integer2; // assign total to sum
```

calculates the total of variables integer1 and integer2, then assigns the result to variable sum using the assignment operator (=). The statement is read as, "sum gets the value of the expression integer1 + integer2." Most calculations are performed in assignments.

Binary Operators

The = operator and the + operator are binary operators—each has two operands. The + operator's operands are integer1 and integer2. The = operator's operands are sum and the value of the expression integer1 + integer2. Place spaces on either side of a binary operator to make the operator stand out and make the program more readable.

Printing with a Format Control String

The format control string "Sum is %d\n" in line 19

```
printf("Sum is %d\n", sum); // print sum
```

contains some literal characters to display ("Sum is ") and the conversion specification %d, which is a placeholder for an integer. The sum is the value to insert in place of %d. The conversion specification for an integer (%d) is the same in both printf and scanf—this is true for most, but not all, C data types.

Combining a Variable Definition and Assignment Statement

You can initialize a variable in its definition. For example, lines 16 and 17 can add the variables integer1 and integer2, then initialize the variable sum with the result:

```
int sum = integer1 + integer2; // assign total to sum
```

Calculations in printf Statements

Actually, we do not need the variable sum, because we can perform the calculation in the printf statement. So, lines 16–19 can be replaced with

```
printf("Sum is %d\n", integer1 + integer2);
```



- I (Multiple Choice) Which statement correctly prompts the user for input?
 - a) printf("Enter the day of the week: ")
 - b) printf(Enter the day of the week:);
 - c) printf('Enter the day of the week: ');
 - d) printf("Enter the day of the week: ");

Answer: d.

2 (*Multiple Choice*) The following statement is read as, "sum *gets* the value of the expression integer1 + integer2." In the statement, = is the ______ operator.

```
sum = integer1 + integer2;
```

- a) equality.
- b) comparison.
- c) assignment.
- d) None of the above.

Answer: c.

2.4 Memory Concepts

Every variable has a name, a type, a value and a location in the computer's memory. In Fig. 2.4's addition program, when line 11

```
scanf("%d", &integer1); // read an integer
```

executes, the program places the user's input into integer1's memory location. Suppose the user enters 45 as integer1's value. Conceptually, memory appears as follows:

When a value is placed in a memory location, it replaces the location's previous value, which is lost. So, this process is said to be **destructive**.

Returning to our addition program again, when line 14

```
scanf("%d", &integer2); // read an integer
```

executes, suppose the user enters 72. Conceptually, memory appears as follows:

integer1	45
integer2	72

These locations are not necessarily adjacent in memory.

Once we have values for integer1 and integer2, line 18

adds these values and places the total into variable sum, replacing its previous value. Conceptually, memory now appears as follows:

integer1	45
integer2	72
sum	117

The integer1 and integer2 values are unchanged by the calculation, which uses but does not destroy the values. Thus, reading a value from a memory location is **nondestructive**.

Self Check

- [(Multiple Choice) Which of the following statements a), b) or c) is false?
 - a) Variable names correspond to locations in the computer's memory.
 - b) Every variable has a name, a type and a value.
 - c) When a value is placed in a memory location, it replaces the previous value in that location. The previous value is lost, so this process is said to be destructive. When a value is read from a memory location, the process is said to be nondestructive.
 - d) All of the above statements are true.

Answer: d.

2.5 Arithmetic in C

Most C programs perform calculations using the following binary arithmetic operators:

C operation	Arithmetic operator	Algebraic expression	C expression
Addition	+	f+7	f + 7
Subtraction	-	p-c	p - c
Multiplication	*	bm	b * m
Division	/	x/y or $\frac{x}{y}$	x / y
Remainder	%	x/y or $\frac{x}{y}$ $r \mod s$	r % s

Note the use of various special symbols not used in algebra. The asterisk (*) indicates multiplication, and the percent sign (%) denotes the remainder operator (introduced below). In algebra, to multiply *a* times *b*, we place these single-letter variable names

side-by-side, as in *ab*. In C, ab would be interpreted as a single, two-letter name (or identifier). Most programming languages denote multiplication by using the * operator, as in a * b.

Integer Division and the Remainder Operator

Integer division (that is, dividing one integer by another) yields an integer result, so 7 / 4 evaluates to 1, and 17 / 5 evaluates to 3. The integer-only remainder operator, %, yields the remainder after integer division, so 7 % 4 yields 3 and 17 % 5 yields 2. We'll discuss several interesting applications of the remainder operator.

An attempt to divide by zero usually is undefined on computer systems. Gener-ERR (x) ally, it results in a fatal error that causes the program to terminate immediately without having successfully performed its job. Nonfatal errors allow programs to run to completion, often producing incorrect results.

Arithmetic Expressions in Straight-Line Form

Arithmetic expressions must be written in straight-line form to facilitate entering programs into a computer. Expressions like "a divided by b" must be written as a/b with all operators and operands in a straight line. The algebraic notation

 $\frac{a}{b}$

is generally not acceptable to compilers, although some special-purpose software packages support more natural notation for complex mathematical expressions.

Parentheses for Grouping Subexpressions

Parentheses are used in C expressions in the same manner as in algebraic expressions. For example, to multiply a times the quantity b + c, we write a * (b + c).

Rules of Operator Precedence

C applies the operators in arithmetic expressions in a precise sequence determined by the following rules of operator precedence, which are generally the same as those in algebra:

1. Expressions grouped in parentheses evaluate first. Parentheses are said to be at the "highest level of precedence." In nested parentheses, such as

$$((a + b) + c)$$

the operators in the innermost pair of parentheses are applied first.

- 2. *, / and % are applied next. If an expression contains several *, / and % operators, evaluation proceeds left-to-right. These three operators are said to be on the same level of precedence.
- 3. + and are evaluated next. If an expression contains + and operators, evaluation proceeds left-to-right. These two operators have the same level of precedence, which is lower than that of *, / and %.
- **4.** The assignment operator (=) is evaluated last.

The operator precedence rules specify the order C uses to evaluate expressions.¹ When we say evaluation proceeds left-to-right, we're referring to the operator's **grouping**, which is sometimes called **associativity**. Some operators group right-to-left.

Sample Algebraic and C Expressions

Let's consider the evaluation of several expressions. Each example lists an algebraic expression and its C equivalent. The following expression calculates the average (arithmetic mean) of five terms:

Algebra:
$$m = \frac{a+b+c+d+e}{5}$$

C: $m = (a + b + c + d + e) / 5;$

In the C statement, parentheses are required to group the additions because division has higher precedence than addition. The entire quantity (a + b + c + d + e) should be divided by 5. If we erroneously omit the parentheses, we obtain a + b + c + d + e / b, which evaluates incorrectly as

$$a+b+c+d+\frac{e}{5}$$

The following expression is the equation of a straight line:

Algebra:
$$y = mx + b$$

C: $y = m * x + b$;

No parentheses are required. Multiplication evaluates first because it has higher precedence than addition.

The following expression contains remainder (%), multiplication, division, addition, subtraction and assignment operations:

Algebra:
$$z = pr \mod q + w/x - y$$

C: $z = p * r % q + w / x - y;$

The circled numbers indicate the order in which C evaluates the operators. The multiplication, remainder and division evaluate first left-to-right (that is, they group left-to-right) because they have higher precedence than addition and subtraction. Next, the addition and subtraction evaluate left-to-right. Finally, the result is assigned to z.

Parentheses "on the Same Level"

Not all expressions with several pairs of parentheses contain nested parentheses. In the following expression, the parentheses are said to be "on the same level":

$$a * (b + c) + c * (d + e)$$

In this case, the parenthesized expressions evaluate left-to-right.

^{1.} We use simple examples to explain expression evaluation order. Subtle issues occur in more complex expressions that you'll encounter later in the book. We'll discuss these issues as they arise.

Evaluation of a Second-Degree Polynomial

To develop a better understanding of the operator precedence rules, let's take a look at how C evaluates a second-degree polynomial.



The circled numbers under the statement indicate the order in which C performs the operations. C does not have an exponentiation operator, so we represent x^2 as x * x. The standard library's pow ("power") function performs exponentiation, as you'll see in Chapter 4.

In the preceding second-degree polynomial, suppose a = 2, b = 3, c = 7 and x = 5. The following diagram illustrates the order in which the operators are applied:

Step 1.
$$y = 2 * 5 * 5 + 3 * 5 + 7$$
; (Leftmost multiplication)
 $2 * 5 \text{ is } 10$
Step 2. $y = 10 * 5 + 3 * 5 + 7$; (Leftmost multiplication)
 $10 * 5 \text{ is } 50$
Step 3. $y = 50 + 3 * 5 + 7$; (Multiplication before addition)
 $3 * 5 \text{ is } 15$
Step 4. $y = 50 + 15 + 7$; (Leftmost addition)
 $50 + 15 \text{ is } 65$
Step 5. $y = 65 + 7$; (Last addition)
 $65 + 7 \text{ is } 72$
Step 6. $y = 72$ (Last operation—place 72 in y)

Using Parentheses for Clarity

As in algebra, it's acceptable to use redundant parentheses to make an expression clearer. So, the preceding statement could be parenthesized as follows:

$$y = (a * x * x) + (b * x) + c;$$

✓ Self Check

- I (*Multiple Choice*) Which, if any, of the following expressions properly does the C calculation "add 3 to the quantity 4 times 5?"
 - a) 3 + 4 * 5
 - b) 3 + (4 * 5)
 - c) (3 + (4 * 5))
 - d) All of the above.

Answer: d.

2 *(Multiple Choice)* Consider the statement:

$$y = a * x * x + b * x + c;$$

Which of the following variations of the preceding statement contain(s) redundant parentheses?

- a) y = (a * x * x + b * x + c);
- b) y = a * (x * x) + (b * x) + c;
- c) y = (a * x * x) + (b * x) + c;
- d) All of the above.

Answer: d.

2.6 Decision Making: Equality and Relational Operators

Executable statements either perform actions like calculations, input and output, or, as you're about to see, make decisions. For example, a program might determine whether a person's grade on an exam is greater than or equal to 60, so it can decide whether to print the message "Congratulations! You passed."

A condition is an expression that can be *true* (that is, the condition is met) or *false* (that is, the condition isn't met). This section introduces the **if statement**, which allows a program to make a decision based on a condition's value. If the condition is *true*, the statement in the **if** statement's body executes; otherwise, it does not.

Equality and Relational Operators

Conditions are formed using the following equality and relational operators:

Algebraic equality or relational operator	C equality or relational operator	Sample C condition	Meaning of C condition		
Relational operators					
>	>	x > y	x is greater than y		
<	<	x < y	x is less than y		
≥	>=	x >= y	x is greater than or equal to y		
≤	<=	x <= y	x is less than or equal to y		
Equality operators					
=	==	x == y	x is equal to y		
≠	!=	x != y	x is not equal to y		

The relational operators <, <=, > and >= have the same precedence and group left-to-right. The equality operators == and != have the same precedence, which is lower than

that of the relational operators, and also group left-to-right. In C, a condition may actually be any expression that generates a zero (*false*) or nonzero (*true*) value.

Confusing the Equality Operator == with the Assignment Operator

ERR
Confusing == with the assignment operator (=) is a common programming error. To avoid this confusion, read the equality operator as "double equals" and the assignment operator as "gets" or "is assigned the value of." As you'll see, confusing these operators can cause difficult-to-find logic errors rather than compilation errors.

Demonstrating the if Statement

Figure 2.5 uses six if statements to compare two numbers entered by the user. For each if statement with a *true* condition, the corresponding printf executes. The program and three sample execution outputs are shown in the figure.

```
I // fig02_05.c
 2 // Using if statements, relational
 3 // operators, and equality operators.
 4 #include <stdio.h>
 6 // function main begins program execution
 7 int main(void) {
       printf("Enter two integers, and I will tell you\n");
 8
 9
       printf("the relationships they satisfy: ");
10
       int number1 = 0; // first number to be read from user
11
12
       int number2 = 0; // second number to be read from user
13
14
       scanf("%d %d", &number1, &number2); // read two integers
15
16
       if (number1 == number2) {
17
          printf("%d is equal to %d\n", number1, number2);
18
       } // end if
19
       if (number1 != number2) {
20
21
          printf("%d is not equal to %d\n", number1, number2);
22
       } // end if
23
       if (number1 < number2) {</pre>
24
25
          printf("%d is less than %d\n", number1, number2);
26
       } // end if
27
28
       if (number1 > number2) {
29
          printf("%d is greater than %d\n", number1, number2);
30
       } // end if
31
32
       if (number1 <= number2) {</pre>
33
          printf("%d is less than or equal to %d\n", number1, number2);
34
       } // end if
35
36
       if (number1 >= number2) {
```

Fig. 2.5 | Using if statements, relational operators, and equality operators. (Part 1 of 2.)

```
printf("%d is greater than or equal to %d\n", number1, number2);
37
38
         } // end if
     } // end function main
Enter two integers, and I will tell you the relationships they satisfy: \bf 3\ 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

```
Enter two integers, and I will tell you the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
```

```
Enter two integers, and I will tell you
  the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

Using if statements, relational operators, and equality operators. (Part 2 of 2.) Fig. 2.5

The program uses scanf (line 14) to read two integers into the int variables number1 and number2. The first %d converts a value to be stored in the variable number1. The second converts a value to be stored in the variable number 2.

Comparing Numbers

The if statement in lines 16–18

```
if (number1 == number2) {
   printf("%d is equal to %d\n", number1, number2);
```

compares number1's and number2's values for equality. If the values are equal, line 17 displays a line of text indicating that the numbers are equal. For each true condition in the if statements starting in lines 20, 24, 28, 32 and 36, the corresponding body statement displays a line of text. Indenting each if statement's body and placing blank lines above and below each if statement enhances program readability.

A left brace, {, begins the body of each if statement (e.g., line 16). A corresponding right brace, }, ends each if statement's body (e.g., line 18). Any number of statements can be placed in an if statement's body.²

Placing a semicolon immediately to the right of the right parenthesis after an if statement's condition is a common error. In this case, the semicolon is treated as an 💢 ERR empty statement that does not perform a task—the statement that was intended to be part of the if statement's body no longer is and always executes.



^{2.} Using braces to delimit an if statement's body is optional for a one-statement body, but it's considered good practice to always use these braces. In Chapter 3, we'll explain the issues.

Operators Introduced So Far

The table below lists from highest-to-lowest precedence the operators introduced so far:

Ope	rators	Grouping		
() * + < == =	/ - <= !=	%	>=	left-to-right left-to-right left-to-right left-to-right left-to-left

The assignment operator (=) groups right-to-left. Refer to the operator precedence chart when writing expressions containing many operators. Confirm that the operators in the expression are applied in the proper order. If you're uncertain about the order of evaluation in a complex expression, use parentheses to group expressions or break the statement into several simpler statements.

Keywords

Some words that we've used in this chapter's examples, such as int, if and void, are keywords or reserved words of the language and have special meaning to the compiler. The following table contains the C keywords. Do not use them as identifiers.

Keywords						
auto	do	goto	signed	unsigned		
break	double	if	sizeof	void		
case	else	int	static	volatile		
char	enum	long	struct	while		
const	extern	register	switch			
continue	float	return	typedef			
default	for	short	union			
Keywords added in the C99 standard _Bool _Complex _Imaginary inline restrict						
Keywords added in the C11 standard						
_Alignas _Align	nof _Atomic _Ge	neric _Noreturn	_Static_assert	_Thread_local		

Self Check

- [Multiple Choice] Which of the following statements is false?
 - a) A condition is an expression that can be *true* or *false*.
 - b) A condition may be any expression that generates a zero (true) or nonzero (false) value.

- c) The if statement makes a decision based on a condition's value. If the condition is *true*, the statement in the if statement's body executes; otherwise, it does not.
- d) You form conditions using the equality operators and relational operators. **Answer:** b) is *false*. Actually, in C, a condition may be any expression that generates a zero (*false*) or nonzero (*true*) value.
- **2** (*Multiple Choice*) Which of the following statements is *false*?
 - a) The following if statement executes its body if number1 equals number2:

```
if (number1 == number2) {
    printf("%d is equal to %d\n", number1, number2);
} // end if
```

- b) If you're uncertain about a complex expression's evaluation order, use parentheses to group expressions or break the statement into simpler statements.
- c) Any number of statements can be placed in the body of an if statement. Using braces to delimit the body of an if statement is required.
- d) Some of C's operators, such as the assignment operator (=), group right-to-left rather than left-to-right.

Answer: c) is *false*. Actually, using braces to delimit the body of an if statement is optional when the body contains only one statement. Nevertheless, always using these braces helps avoid errors.

2.7 Secure C Programming

SEC

We mentioned *SEI CERT C Coding Standard* in the Preface and indicated that we'd follow certain guidelines to help you avoid programming practices that open systems to attacks.

Avoid Single-Argument printfs



One such guideline is to avoid using printf with a single string argument.³ printf's first argument is a format string, which printf inspects for conversion specifications. It then replaces each conversion specification with a subsequent argument's value. It tries to do this regardless of whether there is a subsequent argument to use.

In a later chapter, you'll learn how to input strings from users. Though the first printf argument typically is a string literal, it could be a variable containing a string that was input from a user. In such cases, an attacker can craft a user-input format string with more conversion specifications than there are additional printf arguments. This exploit has been used by attackers to read memory that they should not be able to access.⁴

^{3.} For more information, see CERT rule FIO30-C (https://wiki.sei.cmu.edu/confluence/display/c/FI030-C.+Exclude+user+input+from+format+strings). Chapter 6's Secure C Programming section explains the notion of user input as referred to by this CERT guideline.

^{4. &}quot;Format String Attack," Format String Software Attack | OWASP Foundation. Accessed July 22, 2020, https://owasp.org/www-community/attacks/Format_string_attack.

There are a couple of preventative measures you can take to prevent such an attack. If you need to display a string that terminates with a newline, rather than printf, use the puts function, which displays its string argument followed by a newline. For example, in Fig. 2.1, line 7

```
printf("Welcome to C!\n");
should be written as
      puts("Welcome to C!");
```

Function puts simply displays its string argument's contents, so a conversion specification would be displayed as its individual characters.

To display a string without a terminating newline character, use printf with two arguments—a "%s" format control string and the string to display. The %s conversion specification is a placeholder for a string. For example, in Fig. 2.2, line 7

```
printf("Welcome ");
should be written as
      printf("%s", "Welcome ");
```

As with puts, if printf's second argument contains a conversion specification, it will be displayed as its individual characters.

As written, this chapter's printfs actually are secure, but these changes are responsible coding practices that will eliminate certain security vulnerabilities as we get deeper into C. We'll explain the rationales later in the book. From this point forward, we use these practices in our examples, and you should use them in your code.

scanf, printf, scanf_s and printf_s



We'll be saying more about scanf and printf in subsequent Secure C Programming SEC sections, beginning with Section 3.13. We'll also discuss scanf_s and printf_s, which were introduced in C11 as an attempt to eliminate various scanf and printf security vulnerabilities. In a later Secure C Programming section, we'll discuss wellknown scanf security vulnerabilities and how to avoid them.

Self Check

(Code) Rewrite the following statement as an equivalent secure puts statement:

```
printf("Enter your age:\n");
```

Answer: puts("Enter your age:");

2 (*Code*) Rewrite the following statement as an equivalent secure printf statement:

```
printf("Enter your age:");
```

```
Answer: printf("%s", "Enter your age:");
```

Summary

This chapter introduced many important C features, including displaying data on the screen, inputting data from the user, performing calculations and making decisions. In the next chapter, we build upon these techniques as we introduce structured programming. You'll become more familiar with indentation techniques. We'll study how to specify the *order in which statements are executed*—this is called **flow of control**.

Section 2.1 Introduction

• C facilitates a structured and disciplined approach to computer-program design.

Section 2.2 A Simple C Program: Printing a Line of Text

- Comments (p. 56) begin with //. They document programs (p. 56) and improve program readability. Multi-line comments begin with /* and end with */ (p. 57).
- Comments are ignored by the compiler.
- The **preprocessor** processes lines beginning with # before the program is compiled. The **#include** directive tells the preprocessor (p. 57) to include the contents of another file.
- The **<stdio.h>** header (p. 57) contains information used by the compiler to ensure that you correctly use standard input/output library functions, such as printf.
- The function main is a part of every program. The parentheses after main indicate that main is a program building block called a function (p. 57). Programs contain one or more functions, one of which must be main, which is where the program begins executing.
- Functions can return information. The keyword **int** to the left of main indicates that main "returns" an integer (whole number) value.
- Functions can receive information when they're called upon to execute. The **void** in parentheses after main indicates that main does not receive any information.
- A left brace, {, begins every function's body (p. 57). A corresponding right brace, }, ends each function (p. 57). A pair of braces and the code between them is called a block.
- The **printf function** (p. 58) instructs the computer to display information on the screen.
- A string is sometimes called a character string, a message or a literal (p. 58).
- Every statement (p. 58) must end with the semicolon statement terminator (p. 58).
- In \n (p. 58), the backslash (\) is an **escape character** (p. 58). When encountering a backslash in a string, the compiler combines it with the next character to form an **escape sequence** (p. 58). The escape sequence \n means **newline**.
- When a newline appears in a string output by printf, the output cursor positions to the beginning of the next line.
- The double backslash (\\) escape sequence places a single backslash in a string.
- The escape sequence \" represents a literal double-quote character.

Section 2.3 Another Simple C Program: Adding Two Integers

- A variable (p. 61) is a location in memory where a value can be stored for use by a program.
- Variables of type int (p. 61) hold whole-number integer values.
- All variables must be defined with a name and a type before they can be used in a program.
- A variable name in C is any valid **identifier** (p. 61). An identifier is a series of characters consisting of letters, digits and underscores (_) that does not begin with a digit.
- C is case sensitive (p. 61).
- Function scanf (p. 62) gets input from the standard input—usually the keyboard.
- The **scanf format control string** (p. 62) indicates the type(s) of data to input.

- The **%d** conversion specification (p. 62) indicates an integer (the letter d stands for "decimal integer"). The % begins each conversion specification.
- The arguments that follow scanf's format control string begin with an ampersand (&) followed by a variable name. In this context, the ampersand—called the address operator (p. 62)—tells scanf the variable's memory location. The computer then stores the value at that location.
- Most calculations are performed in assignment statements (p. 63).
- The = operator and the + operator are binary operators—each has two operands (p. 63).
- In a printf that specifies a format control string as its first argument, the conversion specifications indicate placeholders for data to output.

Section 2.4 Memory Concepts

- Every variable has a name, a type, a value and a memory location.
- When a value is placed in a memory location, it replaces the location's previous value, which is lost. So this process is said to be **destructive** (p. 64).
- Reading a value from a memory location is **nondestructive** (p. 65).

Section 2.5 Arithmetic in C

- Most programming languages denote multiplication with the * operator, as in a * b.
- Arithmetic expressions must be written in straight-line form (p. 66) to facilitate entering programs into the computer.
- Parentheses group terms in C expressions in much the same manner as in algebraic expressions.
- C evaluates arithmetic expressions in a precise sequence determined by the following rules of operator precedence (p. 66), which are generally the same as those followed in algebra.
- Expressions containing several +, / and % operations evaluate left-to-right. These three operators are on the same level of precedence.
- Expressions containing several + and operations evaluate left-to-right. These two operators have the same level of precedence, which is lower than that of *, / and %.
- Operator **grouping** (p. 67) specifies whether operators evaluate left-to-right or right-to-left.

Section 2.6 Decision Making: Equality and Relational Operators

- Executable C statements either perform actions or make decisions.
- C's **if statement** (p. 69) allows a program to make a decision based on whether a condition (p. 69) is *true* (p. 69) or *false* (p. 69). If the condition is *true*, the if statement's body executes; otherwise, it does not.
- You form conditions in if statements using the equality and relational operators (p. 69).
- The relational operators all have the same level of precedence and group left-to-right. The
 equality operators have lower precedence than the relational operators and also group leftto-right.
- To avoid confusing assignment (=) and equality (==), the assignment operator should be read "gets," and the equality operator should be read "double equals."
- The compiler usually ignores white-space characters such as tabs, newlines and spaces.
- Keywords (p. 72; or reserved words) have special meaning to the C compiler, so you cannot use them as identifiers such as variable names.

Section 2.7 Secure C Programming

- One practice to help avoid leaving systems open to attacks is to avoid using printf with a single string argument.
- To display a string followed by a newline character, use the **puts function** (p. 74), which displays its string argument followed by a newline character.
- To display a string without a trailing newline character, use printf with the "%s" conversion specification (p. 74) as the first argument and the string to display as the second argument.

Self-Review Exercises

2.1	Fill-In the blanks in each of the following.
	a) Every C program begins execution at the function
	b) Every function's body begins with and ends with
	c) Every statement ends with a(n)
	d) The standard library function displays information on the screen.
	e) The escape sequence \n represents the character, which causes the
	cursor to position to the beginning of the next line on the screen.
	f) The standard library function obtains data from the keyboard.
	g) The conversion specification in a printf or scanf format control
	string indicates that an integer will be output or input, respectively.
	h) Whenever a new value is placed in a memory location, that value overrides
	the previous value in that location. This process is said to be
	i) When a value is read from a memory location, the value in that location is
	preserved; this process is said to be
	j) The statement is used to make decisions.

- **2.2** State whether each of the following is *true* or *false*. If *false*, explain why.
 - a) Function printf always begins printing at the beginning of a new line.
 - b) Comments cause the computer to display the text after // on the screen when the program is executed.
 - c) The escape sequence \n in a printf format control string positions the output cursor to the beginning of the next line.
 - d) All variables must be defined before they're used.
 - e) All variables must be given a type when they're defined.
 - f) C considers the variables number and NuMbEr to be identical.
 - g) Definitions can appear anywhere in the body of a function.
 - h) All arguments following the format control string in a printf function must be preceded by an ampersand (&).
 - i) The remainder operator (%) can be used only with integer operands.
 - j) The arithmetic operators *, /, %, + and all have the same precedence.
 - k) A program that prints three lines of output must contain three printfs.
- **2.3** Write a single C statement to accomplish each of the following:
 - a) Define the variable number to be of type int and initialize it to 0.
 - b) Prompt the user to enter an integer. End your prompting message with a colon (:) followed by a space and leave the cursor positioned after the space.

- c) Read an integer from the keyboard and store the value in integer variable a.
- d) If number is not equal to 7, display "number is not equal to 7."
- e) Display "This is a C program." on one line.
- f) Display "This is a C program." on two lines so the first line ends with C.
- g) Display "This is a C program." with each word on a separate line.
- h) Display "This is a C program." with the words separated by tabs.
- **2.4** Write a statement (or comment) to accomplish each of the following:
 - a) State that a program will calculate the product of three integers.
 - b) Prompt the user to enter three integers.
 - c) Define the variable x to be of type int and initialize it to 0.
 - d) Define the variable y to be of type int and initialize it to 0.
 - e) Define the variable z to be of type int and initialize it to 0.
 - f) Read three integers from the keyboard and store them in variables x, y and z.
 - g) Define the variable result, compute the product of the integers in the variables x, y and z, and use that product to initialize the variable result.
 - h) Display "The product is" followed by the value of the int variable result.
- **2.5** Using the statements you wrote in Exercise 2.4, write a complete program that calculates the product of three integers.
- **2.6** Identify and correct the errors in each of the following statements:

```
a) printf("The value is %d\n", &number);
b) scanf("%d%d", &number1, number2);
c) if (c < 7);{
    puts("C is less than 7");
}
d) if (c => 7) {
    puts("C is greater than or equal to 7");
}
```

Answers to Self-Review Exercises

- **2.1** a) main. b) left brace ({), right brace (}). c) semicolon. d) printf. e) newline. f) scanf. g) %d. h) destructive. i) nondestructive. j) if.
- **2.2** See the answers below:
 - a) False. Function printf always begins printing where the cursor is positioned, and this may be anywhere on a line of the screen.
 - b) False. Comments do not cause any action to be performed when the program is executed. They're used to document programs and improve their readability.
 - c) True.
 - d) True.
 - e) True.
 - f) False. C is case sensitive, so these variables are different.
 - g) True.

- h) *False*. Arguments in a printf function ordinarily should not be preceded by an ampersand. Arguments following the format control string in a scanf function ordinarily should be preceded by an ampersand. We'll discuss exceptions to these rules in Chapter 6 and Chapter 7.
- i) True.
- j) *False*. The operators *, / and % are on the same level of precedence, and the operators + and are on a lower level of precedence.
- k) *False*. A printf statement with multiple \n escape sequences can print several lines.

```
2.3 See the answers below:
```

```
a) int number = 0;
     b) printf("%s", "Enter an integer: ");
     c) scanf("%d", &a);
     d) if (number != 7) {
           puts("The variable number is not equal to 7.");
     e) puts("This is a C program.");
     f) puts("This is a C\nprogram.");
     g) puts("This\nis\na\nC\nprogram.");
     h) puts("This\tis\ta\tC\tprogram.");
     See the answers below:
2.4
     a) // Calculate the product of three integers
     b) printf("%s", "Enter three integers: ");
     c) int x:
     d) int y;
     e) int z;
     f) scanf("%d%d%d", &x, &y, &z);
     g) int result = x * y * z;
     h) printf("The product is %d\n", result);
```

2.5 See below.

```
// Calculate the product of three integers
2
   #include <stdio.h>
3
   int main(void) {
4
       printf("Enter three integers: "); // prompt
5
6
7
       int x = 0;
8
       int y = 0;
9
       int z = 0;
       scanf("%d%d%d", &x, &y, &z); // read three integers
10
\mathbf{H}
       int result = x * y * z; // multiply values
12
       printf("The product is %d\n", result); // display result
13
14 } // end function main
```

- See the answers below: 2.6
 - a) Error: &number.

Correction: Eliminate the &. We discuss exceptions to this later.

- b) Error: number2 does not have an ampersand. Correction: number2 should be &number2. Later in the text, we discuss exceptions to this.
- c) Error: Semicolon after the right parenthesis of the condition in the if statement. The puts will execute whether or not the if statement's condition is true. The semicolon after the right parenthesis is an empty statement that does nothing

Correction: Remove the semicolon after the right parenthesis.

d) Error: => is not an operator in C. Correction: The relational operator => should be changed to >= (greater than or equal to).

Exercises

- **2.7** Identify and correct the errors in each of the following statements. (*Note:* There may be more than one error per statement.)
 - a) scanf("d", value); b) printf("The product of %d and %d is %d"\n, x, y); c) firstNumber + secondNumber = sumOfNumbers d) if (number => largest) { largest == number; e) */ Program to determine the largest of three integers /* f) Scanf("%d", anInteger); g) printf("Remainder of %d divided by %d is\n", x, y, x % y); h) if (x = y); { printf(%d is equal to %d\n", x, y); i) print("The sum is %d\n," x + y); j) Printf("The value you entered is: %d\n, &value);
- Fill-In the blanks in each of the following:
 - a) _____ are used to document a program and improve its readability.
 - b) The function used to display information on the screen is _____.
 - c) A C statement that makes a decision is _____.
 - d) Calculations are normally performed by _____ statements.
 - e) The _____ function inputs values from the keyboard.
- Write a single C statement or line that accomplishes each of the following:
 - a) Display the message "Enter two numbers."
 - b) Assign the product of variables b and c to variable a.

- c) State that a program performs a sample payroll calculation (i.e., use text that helps to document a program).
- d) Input three integer values and place them in int variables a, b and c.
- **2.10** State which of the following are *true* and which are *false*. If *false*, explain why.
 - a) C operators evaluate left-to-right.
 - b) Each of the following is a valid variable name: _under_bar_, m928134, t5, j7, her_sales, his_account_total, a, b, c, z, z2.
 - c) The statement printf("a = 5;"); is an example of an assignment statement.
 - d) An arithmetic expression containing no parentheses evaluates left-to-right.
 - e) The following are all invalid variable names: 3g, 87, 67h2, h22, 2h.
- **2.11** Fill-In the blanks in each of the following:
 - a) What arithmetic operations are on the same level of precedence as multiplication? _____.
 - b) When parentheses are nested, which set of parentheses is evaluated first in an arithmetic expression? _____.
 - c) A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a _____.
- **2.12** What, if anything, displays when each of the following statements is performed? If nothing displays, then answer "Nothing." Assume x = 2 and y = 3.

```
a) printf("%d", x);
```

```
b) printf("%d", x + x);
```

- c) printf("%s", "x=");
- d) printf("x=%d", x);
- e) printf("d = d", x + y, y + x);
- f) z = x + y;
- g) scanf("%d%d", &x, &y);
- (h) // printf("x + y = %d", x + y);
- i) printf("%s", "\n");
- **2.13** Which of the following C statements contain variables whose values are replaced?

```
a) scanf("%d%d%d%d%d", &b, &c, &d, &e, &f);
```

```
b) p = i + j + k + 7;
```

- c) printf("%s", "Values are replaced");
- d) printf("%s", "a = 5");
- **2.14** Given the equation $y = ax^3 + 7$, which of the following, if any, are correct C statements for this equation?

```
a) y = a * x * x * x + 7;
```

b)
$$y = a * x * x * (x + 7);$$

c)
$$y = (a * x) * x * (x + 7);$$

d)
$$y = (a * x) * x * x + 7;$$

e)
$$y = a * (x * x * x) + 7;$$

f) y = a * x * (x * x + 7);

2.15 State the order of evaluation of the operators in each of the following C statements and show the value of x after each statement is performed.

```
a) x = 7 + 3 * 6 / 2 - 1;
b) x = 2 % 2 + 2 * 2 - 2 / 2;
c) x = (3 * 9 * (3 + (9 * 3 / (3))));
```

- **2.16** (Arithmetic) Write a program that reads two integers from the user then displays their sum, product, difference, quotient and remainder.
- **2.17** (*Displaying Values with printf*) Write a program that displays the numbers 1 to 4 on the same line. Write the program using the following methods.
 - a) Using one printf statement with no conversion specifications.
 - b) Using one printf statement with four conversion specifications.
 - c) Using four printf statements.
- **2.18** (Comparing Integers) Write a program that reads two integers from the user then displays the larger number followed by the words "is larger." If the numbers are equal, display the message "These numbers are equal." Use only the single-selection form of the if statement you learned in this chapter.
- **2.19** (Arithmetic, Largest Value and Smallest Value) Write a program that inputs three different integers from the keyboard, then displays the sum, the average, the product, the smallest and the largest of these numbers. Use only the single-selection form of the if statement you learned in this chapter. The screen dialogue should appear as follows:

```
Enter three different integers: 13 27 14
Sum is 54
Average is 18
Product is 4914
Smallest is 13
Largest is 27
```

- **2.20** (Circle Area, Diameter and Circumference) For a circle of radius 2, display the diameter, circumference and area. Use the value 3.14159 for π . Use the following formulas (r is the radius): diameter = 2r, circumference = $2\pi r$ and area = πr^2 . Perform each of these calculations inside the printf statement(s) and use the conversion specification %f. This chapter discussed only integer constants and variables. Chapter 3 will discuss floating-point numbers—that is, values that can have decimal points.
- **2.21** What does the following code display? printf("%s", "*\n**\n***\n***\n");
- **2.22** (Odd or Even) Write a program that reads an integer and determines and displays whether it's odd or even. Use the remainder operator. An even number is a multiple of two. Any multiple of two leaves a remainder of zero when divided by 2.
- **2.23** (*Multiples*) Write a program that reads two integers and determines and displays whether the first is a multiple of the second. Use the remainder operator.

- **2.24** Distinguish between the terms fatal error and nonfatal error. Why might you prefer to experience a fatal error rather than a nonfatal error?
- **2.25** (Integer Value of a Character) Here's a peek ahead. In this chapter, you learned about integers and the type int. C can also represent uppercase letters, lowercase letters and a considerable variety of special symbols. C uses small integers internally to represent each different character. The set of characters a computer uses together with the corresponding integer representations for those characters is called that computer's character set. You can display the integer equivalent of uppercase A, for example, by executing the statement

```
printf("%d", 'A');
```

Write a C program that displays the integer equivalents of some uppercase letters, lowercase letters, digits and special symbols. At a minimum, determine the integer equivalents of the following: A B C a b c 0 1 2 \$ * + / and the space character.

2.26 (Separating Digits in an Integer) Write a program that inputs one five-digit number, separates the number into its individual digits and displays the digits separated from one another by three spaces each. [Hint: Use combinations of integer division and the remainder operation.] For example, if the user types in 42139, the program should display

```
4 2 1 3 9
```

2.27 (*Table of Squares and Cubes*) Using only the techniques you learned in this chapter, write a program that calculates the squares and cubes of the numbers from 0 to 10 and uses tabs to display the following table of values:

```
number
           square
                      cube
           0
                      0
123456789
           1
                      1
           4
9
                      8
                      27
           16
                      64
           25
           36
                      216
           49
                      343
                      512
           64
           81
                      729
10
           100
                      1000
```

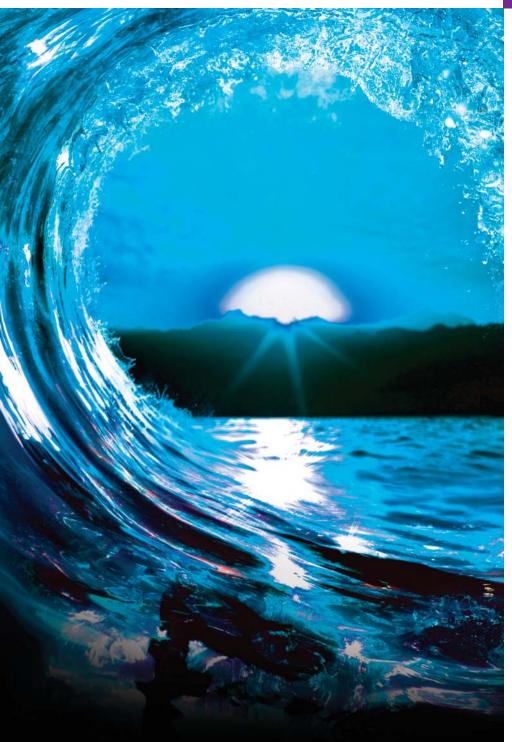
2.28 (Target Heart-Rate Calculator) While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your doctors and trainers. According to the American Heart Association (AHA) (http://bit.ly/AHATargetHeartRates), the formula for calculating your maximum heart rate in beats per minute is 220 minus your age in years. Your target heart rate is 50–85% of your maximum heart rate. Write a program that prompts for and inputs the user's age and calculates and displays the user's maximum heart rate and the range of the user's target heart rate. [These formulas are estimates provided by the AHA; maximum and

84 Chapter 2 Intro to C Programming

target heart rates may vary based on the health, fitness and gender of the individual. Always consult a physician or qualified healthcare professional before beginning or modifying an exercise program.]

2.29 (Sort in Ascending Order) Write a program that inputs three different numbers from the user. Display the numbers in increasing order. Recall that an if statement's body can contain more than one statement. Prove that your script works by running it on all six possible orderings of the numbers. Does your script work with duplicate numbers? [This is challenging. In later chapters you'll do this more conveniently and with many more numbers.]

Structured Program Development



3

Objectives

In this chapter, you'll:

- Use basic problem-solving techniques.
- Develop algorithms through the process of top-down, stepwise refinement.
- Select actions to execute based on a condition using the if and if...else selection statements.
- Execute statements in a program repeatedly using the while iteration statement.
- Use counter-controlled iteration and sentinelcontrolled iteration.
- Use structured programming techniques.
- Use increment, decrement and assignment operators.
- Continue our presentation of Secure C programming.

- 3.1 Introduction
- 3.2 Algorithms
- 3.3 Pseudocode
- 3.4 Control Structures
- **3.5** The **if** Selection Statement
- **3.6** The **if**...**else** Selection Statement
- **3.7** The while Iteration Statement
- **3.8** Formulating Algorithms Case Study 1: Counter-Controlled Iteration

- **3.9** Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Iteration
- **3.10** Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements
- **3.11** Assignment Operators
- **3.12** Increment and Decrement Operators
- **3.13** Secure C Programming

Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises

3.1 Introduction

Before writing a program to solve a problem, you must have a thorough understanding of the problem and a carefully planned solution approach. Chapters 3 and 4 discuss developing structured computer programs. In Section 4.11, we summarize the structured programming techniques developed here and in Chapter 4.

3.2 Algorithms

The solution to any computing problem involves executing a series of actions in a specific order. An **algorithm** is a **procedure** for solving a problem in terms of

- 1. the actions to execute, and
- 2. the order in which these actions should execute.

The following example shows that correctly specifying the order in which the actions should execute is important.

Consider the "rise-and-shine algorithm" followed by one junior executive for getting out of bed and going to work:

- 1. Get out of bed,
- 2. take off pajamas,
- 3. take a shower,
- 4. get dressed,
- 5. eat breakfast, and
- **6.** carpool to work.

This routine gets the executive to work well prepared to make critical decisions. Suppose that the same steps are performed in a slightly different order:

- 1. Get out of bed,
- 2. take off pajamas,
- 3. get dressed,
- 4. take a shower,
- 5. eat breakfast,
- 6. carpool to work.

In this case, our junior executive shows up for work soaking wet. Specifying the order in which statements should execute in a computer program is called **program control**. In this and the next chapter, we investigate C's program control capabilities.

Self Check

I (Fill-in-the-Blank) A procedure for solving a problem in terms of the actions to be executed, and the order in which these actions are to be executed, is called an

Answer: algorithm.

3.3 Pseudocode

Pseudocode is an informal artificial language similar to everyday English that helps you develop algorithms before converting them to structured C programs. Pseudocode is convenient and user friendly. It helps you "think out" a program before writing it in a programming language. Computers do not execute pseudocode.

Pseudocode consists purely of characters, so you may type it in any text editor. Often, converting carefully prepared pseudocode to C is as simple as replacing a pseudocode statement with its C equivalent.

Pseudocode describes the *actions* and *decisions* that will execute once you convert the pseudocode to C and run the program. Definitions are not executable statements—they're simply messages to the compiler. For example, the definition

int
$$i = 0$$
;

tells the compiler variable i's type, instructs the compiler to reserve space in memory for the variable and initializes it to 0. But this definition does not perform an action when the program executes, such as input, output, a calculation or a comparison. So, some programmers do not include definitions in their pseudocode. Others choose to list each variable and briefly mention its purpose.

Self Check

- [(Multiple Choice) Which of the following statements is false?
 - a) Pseudocode is useful for developing algorithms that will be converted to structured C programs.
 - b) Pseudocode is a computer programming language that's more concise than C.

- c) Pseudocode consists purely of characters, so you may conveniently type it in any text-editor program.
- d) Pseudocode describes the actions and decisions that will execute once you convert it to C and run the program.

Answer: b) is *false*. Actually, pseudocode is not an actual computer programming language. It helps you "think out" a program before writing it in a programming language.

3.4 Control Structures

Normally, statements in a program execute one after the other in the order in which you write them. This is called **sequential execution**. As you'll soon see, various C statements enable you to specify that the next statement to execute may be other than the next one in sequence. This is called **transfer of control**.

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of a great deal of difficulty experienced by software-development groups. The finger of blame was pointed at the **goto statement** that allows you to specify a transfer of control to one of many possible destinations in a program. The notion of so-called structured programming became almost synonymous with "**goto elimination**."

The research of Böhm and Jacopini¹ demonstrated that programs could be written without any goto statements. The challenge of the era was for programmers to shift their styles to "goto-less programming." It was not until well into the 1970s that the programming profession started taking structured programming seriously. The results were impressive, as software-development groups reported reduced development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. Programs produced with structured techniques were clearer, easier to debug and modify, and more likely to be bug-free in the first place.

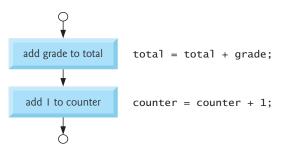
Böhm and Jacopini's work demonstrated that all programs could be written in terms of three control structures, namely the sequence structure, the selection structure and the iteration structure. The sequence structure is simple—unless directed otherwise, the computer executes C statements one after the other in the order in which they're written.

Flowcharts

A flowchart is a graphical representation of an algorithm or of a portion of an algorithm. You draw flowcharts using certain special-purpose symbols such as rectangles, diamonds, rounded rectangles, and small circles, connected by arrows called flowlines.

Flowcharts help you develop and represent algorithms, although pseudocode is preferred by most programmers. Flowcharts clearly show how control structures operate. Consider the following flowchart for a sequence structure in a portion of an algorithm that calculates the class average on a quiz:

^{1.} C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336–371. This classic computer-science paper is available at various online sites.



The rectangle (or action) symbol indicates any action, such as a calculation, input or output. The flowlines indicate the order in which to perform the actions. This program segment first adds grade to total, then adds 1 to counter. As we'll soon see, anywhere in a program a single action may be placed, you may place several actions in sequence.

When drawing a flowchart for a complete algorithm, the first symbol is a rounded rectangle symbol containing "Begin", and the last is a rounded rectangle containing "End". When drawing only a portion of an algorithm, we omit the rounded rectangle symbols in favor of using small circles called **connector symbols**.

Selection Statements in C

C provides three types of selection structures in the form of statements:

- The **if** single-selection statement (Section 3.5) selects (performs) an action (or group of actions) only if a condition is *true*.
- The if...else double-selection statement (Section 3.6) performs one action (or group of actions) if a condition is *true* and a different action (or group of actions) if the condition is *false*.
- The switch multiple-selection statement (discussed in the next chapter) performs one of many different actions, depending on the value of an expression.

Iteration Statements in C

C provides three types of iteration structures in the form of statements, namely while (Section 3.7), do...while, and for. These statements perform tasks repeatedly. We discuss do...while and for in the next chapter.

Summary of Control Statements

That's all there is. C has only seven control statements: sequence, three types of selection and three types of iteration. You form each program by combining as many of each type of control statement as is appropriate for the algorithm the program implements.

We'll see that each control statement's flowchart representation has two small circle symbols, one at the entry point to the control statement and one at the exit point. These single-entry/single-exit control statements make it easy to build clear programs.

We can attach the control-statement flowchart segments to one another by connecting the exit point of one to the entry point of the next. This is similar to a child stacking building blocks, so we call this control-statement stacking. You'll see later in this chapter that the only other way to connect control statements is via nesting.

90

Thus, any C program we'll ever need to build can be constructed from only seven control statements combined in only two ways. This is the essence of simplicity.

✓ Self Check

- I (Multiple Choice) Which of the following statements is false?
 - a) Normally, statements in a program execute one after the other in the order in which they're written. This is called transfer of control.
 - b) Programs can be be written without any goto statements.
 - c) All programs can be written in terms of only three control structures—the sequence structure, the selection structure and the iteration structure.
 - d) The sequence structure is simple—unless directed otherwise, the computer executes C statements one after the other in the order in which they're written.

Answer: a) is false. It's actually called sequential execution.

- **2** (*Multiple Choice*) Which of the following statements is *false*?
 - a) C has only seven control statements: sequence, three types of selection and three types of iteration.
 - b) Single-entry/single-exit control statements make it easy to build clear programs.
 - c) Connecting the exit point of one control statement to the entry point of the next is called control-statement stacking.
 - d) The only other way to connect control statements is via roosting. Thus, any C program we'll ever need to build can be constructed from only seven different types of control statements combined in only two ways.

Answer: d) is *false*. The only other way to connect control statements is via nesting.

3.5 The if Selection Statement

Selection statements choose among alternative courses of action. For example, suppose the passing grade on an exam is 60. The following pseudocode statement determines whether the condition "student's grade is greater than or equal to 60" is *true* or *false*:

```
If student's grade is greater than or equal to 60 Print "Passed"
```

If *true*, then "Passed" is printed, and the next pseudocode statement in order is "performed." Remember that pseudocode isn't a real programming language. If *false*, the printing is ignored, and the next pseudocode statement in order is performed.

The preceding pseudocode is written in C as

```
if (grade >= 60) {
   puts("Passed");
} // end if
```

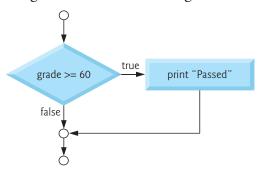
Of course, you'll also need to declare the int variable grade, but the C if statement code corresponds closely to the pseudocode. This is one of the properties of pseudocode that makes it such a useful program-development tool.

Indentation in the if Statement

The indentation in the if statement's second line is optional but highly recommended. It emphasizes the inherent structure of structured programs. The compiler ignores white-space characters such as blanks, tabs and newlines used for indentation and vertical spacing.

if Statement Flowchart

The following flowchart segment illustrates the single-selection if statement:



It contains perhaps the most important flowchart symbol—the diamond (or decision) symbol, which indicates a decision is to be made. The decision symbol's expression typically is a condition that can be *true* or *false*. The two flowlines emerging from it indicate the paths to take when the expression is *true* or *false*. Decisions can be based on any expression's value—zero is *false*, and nonzero is *true*.

The if statement is a single-entry/single-exit statement. We'll soon learn that the flowchart segment for the remaining control structures also can contain rectangle symbols to indicate the actions to be performed and diamond symbols to indicate decisions to be made. This is the action/decision model of programming we've been emphasizing.

Self Check

- [(Multiple Choice) Which of the following statements is false?
 - a) The pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
can be written in C as
    if (grade >= 60) {
        puts("Passed");
    } // end if
```

- b) The two flowlines emerging from the decision flowchart symbol indicate the directions to take when the expression in the symbol is *true* or *false*.
- c) Decisions can be based on any expression—if the expression evaluates to nonzero, it's treated as *false*, and if it evaluates to zero, it's treated as *true*.
- d) The if statement is a single-entry/single-exit statement.

Answer: c) is *false*. Actually, decisions can be based on any expression—if the expression evaluates to zero, it's treated as *false*, and if it evaluates to nonzero, it's treated as *true*.

3.6 The if...else Selection Statement

The if...else selection statement specifies different actions to perform when the condition is true or false. For example, the pseudocode statement

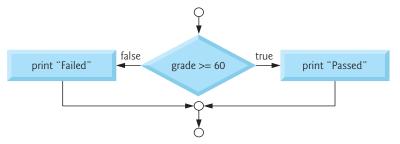
```
If student's grade is greater than or equal to 60
    Print "Passed"
else
   Print "Failed"
```

prints "Passed" if the student's grade is greater than or equal to 60; otherwise, it prints "Failed." In either case, after printing, the next pseudocode statement in sequence "executes." The *else*'s body also is indented. If there are several levels of indentation in a program, each should be indented the same additional amount of space. The preceding pseudocode may be written in C as

```
if (grade >= 60) {
  puts("Passed");
} // end if
else {
   puts("Failed");
} // end else
```

if...else Statement Flowchart

The following flowchart illustrates the if...else statement's flow of control:



Conditional Expressions

The conditional operator (?:) is closely related to the if...else statement. This operator is C's only ternary operator—that is, it takes three operands. A conditional operator and its three operands form a conditional expression. The first operand is a condition. The second is the conditional expression's value if the condition is *true*. The third is the conditional expression's value if the condition is *false*. For example, the conditional-expression argument to the following puts statement evaluates to the string "Passed" if the condition grade >= 60 is true; otherwise, it evaluates to the string "Failed":

```
puts((grade >= 60) ? "Passed" : "Failed");
```

Conditional operators can be used in places where if...else statements cannot, including expressions and arguments to functions (such as printf). Use expressions of the same type for the second and third operands of the conditional operator (?:) ERR (\times) to avoid subtle errors.



Nested if...else Statements

Nested if...else statements test for multiple cases by placing if...else statements inside if...else statements. For example, the following pseudocode statement prints: A for grades greater than or equal to 90, B for grades greater than or equal to 80 (but less than 90), C for grades greater than or equal to 70 (but less than 80), D for grades greater than or equal to 60 (but less than 70), and F for all other grades.

```
If student's grade is greater than or equal to 90
Print "A"

else

If student's grade is greater than or equal to 80
Print "B"
else

If student's grade is greater than or equal to 70
Print "C"
else

If student's grade is greater than or equal to 60
Print "D"
else
Print "F"
```

This pseudocode may be written in C as

```
if (grade >= 90) {
   puts("A");
} // end if
else {
   if (grade >= 80) {
      puts("B");
   } // end if
   else {
      if (grade >= 70) {
         puts("C");
      } // end if
      else {
         if (grade >= 60) {
            puts("D");
         } // end if
         else {
            puts("F");
         } // end else
      } // end else
   } // end else
} // end else
```

If the variable grade is greater than or equal to 90, all four conditions are true, but only the puts statement after the first test executes. Then, the else part of the "outer" if...else statement is skipped, bypassing the rest of the nested if...else statement.

Most programmers write the preceding if statement as

```
if (grade >= 90) {
    puts("A");
} // end if
else if (grade >= 80) {
    puts("B");
} // end else if
else if (grade >= 70) {
    puts("C");
} // end else if
else if (grade >= 60) {
    puts("D");
} // end else if
else {
    puts("F");
} // end else
```

Both forms are equivalent. The latter form avoids the deep indentation to the right, which decreases program readability and sometimes causes lines to wrap.

Blocks and Compound Statements

To include several statements in an if's body, you must enclose the statements in braces ({ and }). A set of statements contained within a pair of braces is called a compound statement or a block. A compound statement can be placed anywhere in a program that a single statement can be placed.

The following if...else statement's else part includes a compound statement containing two statements to execute if the condition is *false*:

```
if (grade >= 60) {
   puts("Passed.");
} // end if
else {
   puts("Failed.");
   puts("You must take this course again.");
} // end else
```

If grade is less than 60, both puts statements in the else execute and the code prints:

```
Failed.
You must take this course again.
```

The braces surrounding the two statements in the else clause are important. Without them, the statement

```
puts("You must take this course again.");
```

would be outside the else's body (and outside the if...else statement) and would execute regardless of whether the grade was less than 60, so even a passing student would have to take the course again. To avoid problems like this, always include your control statements' bodies in braces ({ and }), even if those bodies contain only a single statement. This solves the "dangling-else" problem, which we discuss in this chapter's exercises.

Kinds of Errors

A syntax error (such as misspelling "else") is caught by the compiler. A logic error \(\omega \in \mathbb{R} \) has its effect at execution time. A fatal logic error causes a program to fail and terminate prematurely. A nonfatal logic error allows a program to continue executing but to produce incorrect results.

Empty Statement

Anywhere a single or compound statement can be placed, it's possible to place an empty statement, represented by a semicolon (;). Placing a semicolon after an if's ERR condition, as in

```
if (grade >= 60);
```

leads to a logic error in single-selection if statements and a syntax error in double-selection and nested if...else statements.

Type both braces of compound statements before typing the individual statements within the braces. This helps avoid omission of one or both of the braces, preventing syntax errors (such as an if statement whose if part has multiple statements, which requires a pair of braces) and logic errors. Many integrated development environments and code editors insert the closing brace for you as soon as you type the opening one.

Self Check

I (*TruelFalse*) The following code includes a compound statement in an if...else statement's else part:

```
if (grade >= 60) {
   puts("Passed.");
} // end if
else
   puts("Failed.");
   puts("You must take this course again.");
```

Answer: False. The curly braces around the two puts statements in the else part of this if...else statement are missing. The correct code with the compound statement is

```
if (grade >= 60) {
   puts("Passed.");
} // end if
else {
   puts("Failed.");
   puts("You must take this course again.");
} // end else
```

2 (*True/False*) The conditional expression argument to the following puts statement

```
puts((grade >= 60) : "Passed" ? "Failed");
```

evaluates to the string "Passed" if the condition grade >= 60 is *true*; otherwise, it evaluates to the string "Failed".

Answer: False. This statement won't compile because the conditional operator's ? and: are reversed. The correct statement to produce the desired result is

```
puts((grade >= 60) ? "Passed" : "Failed");
```

3.7 The while Iteration Statement

An iteration statement (also called a repetition statement or loop) repeats an action while some condition remains true. The pseudocode statement

While there are more items on my shopping list Purchase next item and cross it off my list

describes the iteration that occurs during a shopping trip. The condition "there are more items on my shopping list" may be true or false. If it's true, the shopper performs the action "Purchase next item and cross it off my list" repeatedly while the condition remains true. Eventually, the condition will become false (when the last item on the shopping list has been purchased and crossed off the list). At this point, the iteration terminates, and the first pseudocode statement after the iteration statement "executes."

Calculating the First Power of 3 Greater Than 100

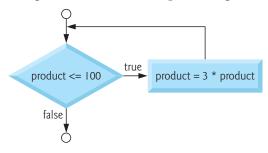
As a while statement example, consider a program segment that finds the first power of 3 larger than 100. The integer variable product is initialized to 3. When the following code segment finishes executing, product will contain the desired answer:

```
int product = 3;
while (product <= 100) {</pre>
   product = 3 * product;
```

The loop repeatedly multiplies product by 3, so it takes on the values 9, 27 and 81 successively. When product becomes 243, the condition product <= 100 becomes false, terminating the iteration—product's final value is 243. Execution continues with the next statement after the while. An action in the while statement's body must eventually cause the condition to become *false*; otherwise, the loop will never termi-ERR (x) nate—a logic error called an infinite loop. The statement(s) contained in a while iteration statement constitute its body, which may be a single statement or a compound statement.

while Statement Flowchart

The following flowchart segment illustrates the preceding while iteration statement:



The flowchart clearly shows the iteration—the flowline emerging from the rectangle points back to the flowline entering the decision. The loop tests the condition in the diamond during each iteration until the condition eventually becomes *false*. At this point, the while statement exits and control continues with the next statement in sequence.



Self Check

I (*Program Segment*) The while statement program segment in this section finds the first power of 3 larger than 100. Rewrite this program segment so that it will find the first power of 2 greater than or equal to 1024, leaving it in product.

Answer: See below.

```
int product = 2;
while (product < 1024) {
   product = 2 * product;
}</pre>
```

2 (*Fill-in-the-Blank*) An action in a while statement's body must eventually cause the condition to become *false*; otherwise, the loop will never terminate. This is a logic error called a(n) _____.

Answer: infinite loop.

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Iteration

To illustrate how algorithms are developed, we solve two variations of a class-averaging problem in this section and the next. Consider the following problem statement:

A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.

The class average is the sum of the grades divided by the number of students. The algorithm to solve this problem must input the grades, then calculate and display the class average.

Pseudocode for the Class-Average Problem

Let's use pseudocode to list the actions to execute and specify the order in which they should execute. We use counter-controlled iteration to input the grades one at a time. This technique uses a variable called a counter to specify the number of times a set of statements should execute. In this example, we know that ten students took a quiz, so we need to input 10 grades. Iteration terminates when the counter exceeds 10. In this case study, we simply present the final pseudocode algorithm (Fig. 3.1) and the corresponding C program (Fig. 3.2). In the next case study, we show how to develop pseudocode algorithms. Counter-controlled iteration is often called definite iteration because the number of iterations is known before the loop begins executing.

```
Set total to zero
Set grade counter to one
While grade counter is less than or equal to ten
Input the next grade
Add the grade into the total
Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Fig. 3.1 | Pseudocode algorithm that uses counter-controlled iteration to solve the class-average problem.

```
| // fig03_02.c
   // Class average program with counter-controlled iteration.
 3 #include <stdio.h>
 4
 5
    // function main begins program execution
   int main(void) {
 7
       // initialization phase
       int total = 0; // initialize total of grades to 0
 8
 9
       int counter = 1; // number of the grade to be entered next
10
       // processing phase
11
12
       while (counter <= 10) { // loop 10 times
          printf("%s", "Enter grade: "); // prompt for input
13
          int grade = 0; // grade value
14
15
          scanf("%d", &grade); // read grade from user
          total = total + grade; // add grade to total
16
17
          counter = counter + 1; // increment counter
       } // end while
18
19
20
       // termination phase
       int average = total / 10; // integer division
21
22
       printf("Class average is %d\n", average); // display result
    } // end function main
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Fig. 3.2 Class-average problem with counter-controlled iteration.

A total is a variable (line 8) used to accumulate the sum of a series of values. A counter is a variable (line 9) used to count—in this case, to count the number of grades entered. Variables for totals should be initialized to zero; otherwise, the sum would include the previous value stored in the total's memory location. You should initialize all counters and totals. Counters typically are initialized to zero or one, depending on their use—we'll present examples of each. An uninitialized variable contains a "garbage" value—the value last stored in the memory location reserved for that variable. If a counter or total isn't initialized, the results of your program will probably be incorrect. These are examples of logic errors.



The average was 81 in the preceding sample execution, but the sum of the grades we input was 817. Of course, 817 divided by 10 should yield 81.7—a number with a decimal point. The next section shows how to deal with such *floating-point numbers*.

\checkmark

Self Check

- I (Fill-in-the-Blank) Counter-controlled iteration is often called ______ iteration because the number of iterations is known before the loop begins executing.

 Answer: definite.
- **2** (*True/False*) Variables used to store totals should be initialized to one before being used in a program; otherwise, the sum would include the previous value stored in the total's memory location.

Answer: False. Actually, Variables used to store totals should be initialized to zero before being used in a program; otherwise, the sum would include the previous value stored in the total's memory location.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Iteration

Let's generalize the class-average problem. Consider the following problem:

Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.

In the first class-average example, we knew there were 10 grades in advance. In this example, no indication is given of how many grades the user might input. The program must process an *arbitrary* number of grades. How can the program determine when to stop inputting grades? How will it know when to calculate and print the class average?

Sentinel Values

One way is to use a sentinel value to indicate "end of data entry." A sentinel value also is called a signal value, a dummy value, or a flag value. The user types grades until all legitimate grades have been entered. The user then types the sentinel value to indicate "the last grade has been entered." Sentinel-controlled iteration is often called indefinite iteration because the number of iterations isn't known before the loop begins executing.

You should choose a sentinel value that cannot be confused with an acceptable input value. Grades on a quiz are non-negative integers, so –1 is an acceptable sentinel value for this problem. Thus, a run of the class-average program might process a stream of inputs such as 95, 96, 75, 74, 89 and –1. The program would then compute and print the class average for the grades 95, 96, 75, 74, and 89. The sentinel value –1 should not enter into the averaging calculation.

Top-Down, Stepwise Refinement

We approach the class-average program with a technique called **top-down**, **stepwise refinement**, which is essential to developing well-structured programs. We begin with a pseudocode representation of the **top**:

Determine the class average for the quiz

The top is a single statement that conveys the program's overall function. As such, the top is, in effect, a complete representation of a program. Unfortunately, the top rarely conveys a sufficient amount of detail for writing the C program. So we now begin the refinement process. We divide the top into smaller tasks listed in the order in which they need to be performed. This results in the following first refinement:

Initialize variables Input, sum, and count the quiz grades Calculate and print the class average

Here, only the sequence structure has been used—the steps listed should execute in order, one after the other. Each refinement, as well as the top itself, is a complete specification of the algorithm. Only the level of detail varies.

Second Refinement

To proceed to the next level of refinement, i.e., the second refinement, we commit to specific variables. We need:

- a running total of the grades,
- a count of how many grades have been processed,
- a variable to receive the value of each grade as it is input and
- a variable to hold the calculated average.

The pseudocode statement

Initialize variables

can be refined as follows:

Initialize total to zero
Initialize counter to zero

Only the total and counter need to be initialized. The variables for the calculated average and the grade the user inputs need not be initialized because their values will be calculated and input from the user, respectively. The pseudocode statement

Input, sum, and count the quiz grades

requires an *iteration structure* that successively inputs each grade. Because we do not know how many grades are to be processed, we'll use sentinel-controlled iteration. The user will enter legitimate grades one at a time. After entering the last legitimate grade, the user will type the sentinel value. The program will test for this value after each grade is input and will terminate the loop when the sentinel is entered. The refinement of the preceding pseudocode statement is then

```
Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
Add this grade into the running total
Add one to the grade counter
Input the next grade (possibly the sentinel)
```

In pseudocode, we do not use braces around the set of statements that form a loop's body. We simply indent the body statements under the *while*. Again, pseudocode is an informal program-development aid.

```
The pseudocode statement
```

Calculate and print the class average

may be refined as follows:

```
If the counter is not equal to zero

Set the average to the total divided by the counter

Print the average
else

Print "No grades were entered"
```

We're being careful here to test for the possibility of division by zero—a fatal error (X) ERR that, if undetected, would cause the program to fail (often called "crashing"). You should explicitly test for this case and handle it appropriately in your program, such as by printing an error message, rather than allowing the fatal error to occur.

Complete Second Refinement

The complete second refinement is shown in Fig. 3.3. We include some blank lines in the pseudocode for readability.

```
Initialize total to zero
Initialize counter to zero
Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
Add this grade into the running total
Add one to the grade counter
Input the next grade (possibly the sentinel)
```

Fig. 3.3 | Pseudocode algorithm that uses sentinel-controlled iteration to solve the class-average problem. (Part 1 of 2.)

```
10 If the counter is not equal to zero
11 Set the average to the total divided by the counter
12 Print the average
13 else
14 Print "No grades were entered"
```

Fig. 3.3 | Pseudocode algorithm that uses sentinel-controlled iteration to solve the class-average problem. (Part 2 of 2.)

Phases in a Basic Program

Many programs can be divided logically into three phases:

- an initialization phase that initializes the program variables,
- a processing phase that inputs data values and adjusts program variables accordingly, and
- a termination phase that calculates and prints the final results.

Number of Pseudocode Refinements

The pseudocode algorithm in Fig. 3.3 solves the more general class-average problem. This algorithm was developed after only two levels of refinement. Sometimes more levels are necessary. You terminate the top-down, stepwise refinement process when the pseudocode algorithm provides sufficient detail for you to convert the pseudocode to C.

The most challenging part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, producing a working C program usually is straightforward. Many programmers write programs without ever using program-development tools such as pseudocode. They feel their ultimate goal is to solve the problem and that writing pseudocode merely delays producing final outputs. This may work for small programs you develop for your own use. But for the substantial programs and software systems you'll likely work on in industry, a formal development process is essential.

Class-Average Program for an Arbitrary Number of Grades

Figure 3.4 shows the C program and two sample executions. Although only integer grades are entered, the averaging calculation is likely to produce a number with a decimal point. The type int cannot represent such a number. So this program introduces the data type double to handle numbers with decimal points—that is, floating-point numbers. We introduce a cast operator to force the averaging calculation to use floating-point numbers. These features are explained after the program listing. Note that lines 13 and 23 both include the sentinel value in the prompts requesting data entry. This is a good practice in a sentinel-controlled loop.

```
// fig03_04.c
    // Class-average program with sentinel-controlled iteration.
 3
    #include <stdio.h>
 4
 5
    // function main begins program execution
 6
    int main(void) {
 7
        // initialization phase
 8
        int total = 0; // initialize total
 9
        int counter = 0; // initialize loop counter
10
II
        // processing phase
12
        // get first grade from user
13
        printf("%s", "Enter grade, -1 to end: "); // prompt for input
14
        int grade = 0; // grade value
15
        scanf("%d", &grade); // read grade from user
16
17
        // loop while sentinel value not yet read from user
18
        while (grade !=-1) {
19
           total = total + grade; // add grade to total
20
           counter = counter + 1; // increment counter
21
22
           // get next grade from user
           printf("%s", "Enter grade, -1 to end: "); // prompt for input
23
24
           scanf("%d", &grade); // read next grade
25
        } // end while
26
27
       // termination phase
        // if user entered at least one grade
28
29
        if (counter != 0) {
30
31
           // calculate average of all grades entered
32
           double average = (double) total / counter; // avoid truncation
33
34
           // display average with two digits of precision
35
           printf("Class average is %.2f\n", average);
36
        } // end if
37
        else { // if no grades were entered, output message
38
           puts("No grades were entered");
39
        } // end else
   } // end function main
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70 Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89 Enter grade, -1 to end: -1
Class average is 82.50
Enter grade, -1 to end: -1
No grades were entered
```

Fig. 3.4 | Class-average program with sentinel-controlled iteration.

Always Use Braces in a while Statement

Without this while loop's braces (lines 18 and 25), only the statement on line 19 would be in the loop's body. The code would be incorrectly interpreted as

```
while (grade != -1)
   total = total + grade; // add grade to total
counter = counter + 1; // increment counter
// get next grade from user
printf("%s", "Enter grade, -1 to end: "); // prompt for input
scanf("%d", &grade); // read next grade
```

ERR (X) This would cause an infinite loop if the user did not input -1 as the first grade.

Converting Between Types Explicitly and Implicitly

Averages often are values such as 7.2 or –93.5 that contain fractional parts. These floating-point numbers can be represented by the data type double. Line 32 defines the variable average as type double to capture the fractional result of our calculation. Normally, the result of the calculation total / counter (line 32) is an integer because total and counter are both int variables. Dividing two ints results in integer division—any fractional part of the calculation is truncated (that is, lost). You can produce a floating-point calculation with integer values by first creating temporary floating-point numbers. C provides the unary cast operator to accomplish this task. Line 32

```
double average = (double) total / counter;
```

uses the cast operator (double) to create a *temporary* floating-point copy of its operand, total. The value stored in total is still an integer. Using a cast operator in this manner is called **explicit conversion**. The calculation now consists of a floating-point value—the temporary double version of total—divided by the int value stored in counter.

C requires the operand data types in arithmetic expressions only to be identical. In mixed-type expressions, the compiler performs an operation called **implicit conversion** on selected operands to ensure that they're of the same type. For example, in an expression containing the data types int and double, copies of int operands are made and implicitly converted to type double. After we explicitly convert total to a double, the compiler implicitly makes a double copy of counter, then performs floating-point division and assigns the floating-point result to average. Chapter 5 discusses C's rules for converting operands of different types.

Cast operators are formed by placing parentheses around a type name. A cast is a unary operator that takes only one operand. C also supports unary versions of the plus (+) and minus (-) operators, so you can write expressions such as -7 or +5. Cast operators group right-to-left and have the same precedence as other unary operators such as unary + and unary -. This precedence is one level higher than that of the multiplicative operators *, / and %.

Formatting Floating-Point Numbers

Figure 3.4 uses the printf conversion specification %.2f (line 35) to format average's value. The f specifies that a floating-point value will be printed. The .2 is the preci-

sion—the value will have two (2) digits to the right of the decimal point. If the %f conversion specification is used without specifying the precision, the default precision is 6 digits to the right of the decimal point, as if the conversion specification %.6f had been used. When floating-point values are printed with precision, the printed value is rounded to the indicated number of decimal positions. The value in memory is unaltered. The following statements display the values 3.45 and 3.4, respectively:

```
printf("%.2f\n", 3.446); // displays 3.45
printf("%.1f\n", 3.446); // displays 3.4
```

Notes on Floating-Point Numbers

Although floating-point numbers are not always "100% precise," they have numerous applications. For example, when we speak of a "normal" body temperature of 98.6 degrees Fahrenheit, we do not need to be precise to a large number of digits. When we view the temperature on a thermometer and read it as 98.6, it may actually be 98.5999473210643. The point here is that calling this number simply 98.6 is fine for most applications. We'll say more about this issue later.

Floating-point numbers often develop through division. When we divide 10 by 3, the result is 3.3333333... with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so the stored floating-point value can be only an approximation. Using floating-point numbers in a manner that assumes they're represented precisely can lead to incorrect results. Floating-point numbers are represented only approximately by most computers. For this reason, you also should not compare floating-point values for equality.



Self Check

- I (*Fill-in-the-Blank*) Sentinel-controlled iteration is often called ______ iteration because the number of iterations isn't known before the loop begins executing.

 Answer: indefinite.
- **2** (Multiple Choice) Which of the following statements is false?
 - a) Many programs can be divided logically into three phases: an initialization phase that initializes the program variables, a processing phase that inputs data values and adjusts program variables accordingly, and a termination phase that calculates and prints the final results.
 - b) You terminate top-down, stepwise refinement when the pseudocode algorithm is specified in sufficient detail for you to be able to convert the pseudocode to C.
 - c) Experience has shown that the most difficult part of solving a problem on a computer is producing a working C program from the algorithm.
 - d) Many programmers write programs without ever using program-development tools such as pseudocode. They feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs.

Answer: c) is *false*. Actually, experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working C program is normally straightforward.

3 (*Program Segment*) When floating-point values are printed with precision, the printed value is rounded. Write statements that display the value 98.5999473210643 with one, four and ten digits of precision, respectively, and specify what's displayed in each case.

Answer: See below.

```
printf("%.1f\n", 98.5999473210643); // displays 98.6
printf("%.4f\n", 98.5999473210643); // displays 98.5999
printf("%.10f\n", 98.5999473210643); // displays 98.5999473211
```

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements

Let's work another complete problem. We'll formulate the algorithm using pseudo-code and top-down, stepwise refinement, and write a corresponding C program. We've seen that control statements may be stacked on top of one another (in sequence) just as a child stacks building blocks. In this case study, we'll see the only other structured way control statements may be connected in C, namely by nesting one control statement within another. Consider the following problem statement:

A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, 10 of the students who completed this course took the licensing examination. Naturally, the college wants to know how well its students did on the exam. You've been asked to write a program to summarize the results. You've been given a list of these 10 students. Next to each name is a 1 if the student passed the exam or a 2 if the student failed.

Your program should analyze the results of the exam as follows:

- 1. Input each test result (i.e., a 1 or a 2). Display the prompting message "Enter result" each time the program requests another test result.
- 2. Count the number of test results of each type.
- 3. Display a summary of the test results indicating the number of students who passed and the number who failed.
- 4. If more than eight students passed the exam, print the message "Bonus to instructor!"

After reading the problem statement carefully, we make the following observations:

- 1. The program must process 10 test results. We'll use a counter-controlled loop.
- 2. Each test result is a number—either a 1 or a 2. Each time the program reads a test result, it must determine whether the result is a 1 or a 2. We'll test for a

1 in our algorithm. If the number is not a 1, we'll assume that it's a 2. Exercise 3.27 asks you to ensure that every test result is a 1 or a 2.

- **3.** Two counters are used—one to count the number of students who passed the exam and one to count the number of students who failed the exam.
- **4.** After the program has processed all the results, it must decide whether more than 8 students passed the exam and, if so, print "Bonus to Instructor!".

Pseudocode Representation of the Top

Let's proceed with top-down, stepwise refinement. We begin with a pseudocode representation of the top:

Analyze exam results and decide whether instructor should receive a bonus

Once again, it's important to emphasize that the top is a complete representation of the program, but multiple refinements are likely to be needed before the pseudocode can be naturally evolved into a C program.

First Refinement

Our first refinement is:

Initialize variables

Input the ten quiz grades and count passes and failures

Print an exam-results summary and decide whether to bonus the instructor

Here, too, even though we have a *complete* representation of the entire program, further refinement is necessary.

Second Refinement

We now commit to specific variables. We need counters to record the passes and failures, a counter to control the looping process, and a variable to store the user input. The pseudocode statement

Initialize variables

can be refined as follows:

Initialize passes to zero Initialize failures to zero Initialize student to one

Only the counter and totals are initialized. The pseudocode statement

Input the ten quiz grades and count passes and failures

requires a loop that successively inputs the result of each exam. Here we know in advance that there are precisely ten exam results, so counter-controlled looping is appropriate. Inside the loop (that is, nested within the loop), a double-selection statement will determine whether each exam result is a pass or a failure and will increment the appropriate counter. The refinement of the preceding pseudocode statement is then

```
While student counter is less than or equal to ten
Input the next exam result
If the student passed
Add one to passes
else
Add one to failures
Add one to student counter
```

The pseudocode statement

Print an exam-results summary and decide whether to bonus the instructor may be refined as follows:

```
Print the number of passes
Print the number of failures
If more than eight students passed
Print "Bonus to instructor!"
```

Complete Second Refinement

Figure 3.5 contains the complete second refinement. We use blank lines for readability.

```
I Initialize passes to zero
    Initialize failures to zero
    Initialize student to one
 3
 4
     While student counter is less than or equal to ten
 5
 6
         Input the next exam result
 7
         If the student passed
 8
             Add one to passes
10
         else
             Add one to failures
11
12
         Add one to student counter
13
    Print the number of passes
15
     Print the number of failures
     If more than eight students passed
18
         Print "Bonus to instructor!"
```

Fig. 3.5 Pseudocode for examination-results problem.

Implementing the Algorithm

This pseudocode is now sufficiently refined for conversion to C. Figure 3.6 shows the C program and two sample executions.

```
// fig03_06.c
    // Analysis of examination results.
 3
   #include <stdio.h>
 4
 5
    // function main begins program execution
    int main(void) {
       // initialize variables in definitions
 7
 8
       int passes = 0;
 9
       int failures = 0;
10
       int student = 1;
11
12
       // process 10 students using counter-controlled loop
13
       while (student <= 10) {</pre>
14
           // prompt user for input and obtain value from user
           printf("%s", "Enter result (1=pass,2=fail): ");
15
16
           int result = 0; // one exam result
           scanf("%d", &result);
17
18
           // if result 1, increment passes
19
20
           if (result == 1) {
21
              passes = passes + 1;
22
           } // end if
23
           else { // otherwise, increment failures
24
              failures = failures + 1;
25
           } // end else
26
27
           student = student + 1; // increment student counter
28
       } // end while
29
30
       // termination phase; display number of passes and failures
       printf("Passed %d\n", passes);
31
32
       printf("Failed %d\n", failures);
33
       // if more than eight students passed, print "Bonus to instructor!"
34
35
       if (passes > 8) {
           puts("Bonus to instructor!");
36
37
       } // end if
    } // end function main
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 2
Enter Result (1=pass, 2=fail): 2
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 2
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 2
Passed 6
Failed 4
```

Fig. 3.6 | Analysis of examination results. (Part 1 of 2.)

```
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 2
Enter Result (1=pass, 2=fail): 1
Passed 9
Failed 1
Bonus to instructor!
```

Fig. 3.6 Analysis of examination results. (Part 2 of 2.)



Self Check

I *(Fill-in-the-Blank)* Control statements may be stacked on top of one another (in sequence) just as a child stacks building blocks. The only other structured way control statements may be connected in C is _______—that is, placing a control statement inside another control statement.

Answer: nesting.

3.11 Assignment Operators

C provides several assignment operators for abbreviating assignment expressions. For example, the statement

```
c = c + 3;
```

can be abbreviated with the addition assignment operator += as

```
c += 3;
```

The += operator adds the value of the expression on the operator's right to the value of the variable on the operator's left then stores the result in the variable on the left. So, the assignment c += 3 adds 3 to c's current value. The following table shows the arithmetic assignment operators, sample expressions using these operators, and explanations:

Assignment operator	Sample expression	Explanation	Assigns
Assume: int $c = 3$, $d =$	5, e = 4, f = 6, g =	= 12;	
+=	c += 7	c = c + 7	10 to c
-=	d -= 4	d = d - 4	1 to d
*=	e *= 5	e = e * 5	20 to e
/=	f /= 3	f = f / 3	2 to f
%=	g %= 9	g = g % 9	3 to g

Self Check

```
1 (Fill-in-the-Blank) The statement
b = b * 5;
can be abbreviated with the multiplication assignment operator *= as ______.
Answer: b *= 5;
2 (Fill-in-the-Blank) What does the assignment c -= 3; do? ______
```

Answer: Subtracts 3 from c's current value.

3.12 Increment and Decrement Operators

The unary increment operator (++) and the unary decrement operator (--) add one to and subtract one from an integer variable, respectively. The following table summarizes the two versions of each operator:

Operator	Sample expression	Explanation
++	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
	b	Decrement b by 1, then use the new value of b in the expression in which b resides.
	b	Use the current value of b in the expression in which b resides, then decrement b by 1.

To increment the variable c by 1, you can use the operator ++ rather than the expressions c = c + 1 or c += 1. If you place ++ or -- before a variable (i.e., prefixed), they're referred to as the **preincrement** or **predecrement operators**. If you place ++ or -- after a variable (i.e., postfixed), they're referred to as the **postincrement** or **postdecrement operators**. By convention, unary operators should be placed next to their operands with no intervening spaces.

Figure 3.7 demonstrates the difference between the preincrementing and the postincrementing versions of the ++ operator. Postincrementing the variable c causes it to be incremented *after* it's used in the printf statement. Preincrementing the variable c causes it to be incremented *before* it's used in the printf statement. The program displays the value of c before and after using ++. The decrement operator (--) works similarly.

```
1  // fig03_07.c
2  // Preincrementing and postincrementing.
3  #include <stdio.h>
```

Fig. 3.7 | Preincrementing and postincrementing. (Part 1 of 2.)

```
5
    // function main begins program execution
   int main(void) {
7
       // demonstrate postincrement
8
       int c = 5; // assign 5 to c
       printf("%d\n", c); // print 5
9
       printf("%d\n", c++); // print 5 then postincrement
10
       printf("%d\n\n", c); // print 6
11
12
13
       // demonstrate preincrement
14
       c = 5; // assign 5 to c
       printf("%d\n", c); // print 5
15
       printf("%d\n", \frac{++c}{}); // preincrement then print 6
16
       printf("%d\n", c); // print 6
17
    } // end function main
5
5
6
5
6
6
```

Fig. 3.7 Preincrementing and postincrementing. (Part 2 of 2.)

```
The three assignment statements in Fig. 3.6
      passes = passes + 1;
      failures = failures + 1;
      student = student + 1;
can be written more concisely with assignment operators as
      passes += 1;
      failures += 1;
      student += 1;
with preincrement operators as
      ++passes;
      ++failures;
      ++student;
or with postincrement operators as
      passes++;
      failures++;
      student++;
```

When incrementing or decrementing a variable in a statement by itself, the preincrement and postincrement forms have the same effect. It's only when a variable appears in the context of a larger expression that preincrementing and postincrementing have different effects (and similarly for predecrementing and postdecrementing).

Only a simple variable name may be used as a ++ or -- operator's operand. Attempting to use the increment or decrement operator on an expression other than a simple variable name is a syntax error—e.g., ++(x+1).



C generally does not specify the order in which an operator's operands will evaluate. We'll see exceptions to this for a few operators in the next chapter. To avoid subtle errors, the ++ and -- operators should be used only in statements that modify exactly one variable.

The following table lists in decreasing precedence order the operators shown so far.

Operators	Grouping	Туре
++ (postfix) (postfix) + - (type) ++ (prefix) (prefix)	right to left right to left	postfix unary
* / %	left to right	multiplicative
+ - < <= > >=	left to right left to right	additive relational
== != ?:	left to right right to left	equality conditional
= += -= *= /= %=	right to left	assignment

The third column names the various groups of operators. Notice that the conditional operator (?:), the unary operators increment (++), decrement (--), plus (+), minus (-) and casts, and the assignment operators =, +=, -=, *=, /= and %= group right-to-left. The other operators group left-to-right.

Self Check

I (Multiple Choice) Given the following code:

--i;

which of the following statements describes what this code does?

- a) Increment i by 1, then use the new value of i in the expression in which i resides.
- b) Use i's current value in the expression in which i resides, then increment i by 1.
- c) Decrement i by 1, then use the new value of i in the expression in which i resides.
- d) Use i's current value in the expression in which i resides, then decrement i by 1.

Answer: c.

2 (What Does This Code Do?) What does this program display as x's final value?

```
#include <stdio.h>

int main(void) {
   int x = 7;
   printf("%d\n", x);
   printf("%d\n", x++);
   printf("%d\n\n", x);
```

Answer: 9.

- 3 (Multiple Choice) Which of the following expressions contains a syntax error?
 - a) ++x + 1
 - b) x++ + x
 - c) ++(x) + 1
 - d) ++(x + 1)

Answer: d. Only a simple variable name may be used as the operand of a ++ or -- operator. Attempting to use the increment or decrement operator on an expression other than a simple variable name is a syntax error—e.g., ++(x + 1).

3.13 Secure C Programming



Arithmetic Overflow

Figure 2.4 presented an addition program that calculated the sum of two int values with the statement

```
sum = integer1 + integer2; // assign total to sum
```

Even this simple statement has a potential problem. Adding the integers could result (X) ERR in a value too large to store in the int variable sum. This is known as arithmetic overflow and can cause undefined behavior, possibly leaving a system open to attack.

The constants INT_MAX and INT_MIN represent the platform-specific maximum and minimum values that can be stored in an int variable. These constants are defined in the header <1imits.h>. There are similar constants for the other integral types that we'll introduce in the next chapter. You can see your platform's values for these constants by opening the header <1imits.h> in a text editor.²

It's good practice to ensure that before you perform arithmetic calculations like the one above, they will not overflow. For an example, see the CERT website.

```
https://wiki.sei.cmu.edu/confluence/display/c/
```

Search for guideline INT32-C. The code uses the && (logical AND) and || (logical OR) operators, which we discuss in Chapter 4. In industrial-strength code, you should perform checks like these for all calculations. Later chapters show other programming techniques for handling such errors.

scanf_s and printf_s

The C11 standard's Annex K introduced more secure versions of printf and scanf called printf_s and scanf_s. We discuss these functions and the corresponding secu-

^{2.} Use your system's search feature to locate the file limits.h.

rity issues in Sections 6.13 and 7.13. Annex K is designated as optional, so not every C vendor implements it. In particular, the GNU C++ and Clang C++ compilers do not implement Annex K, so using scanf_s and printf_s might compromise your code's portability among compilers.

Microsoft implemented its own Visual C++ versions of printf_s and scanf_s before the C11 standard, and its compiler immediately began issuing warnings for every scanf call. The warnings said that scanf was deprecated—it should no longer be used—and that you should consider using scanf_s instead. Microsoft now treats what used to be a warning about scanf as an error. A program with scanf will not compile on Visual C++ and you will not be able to execute the program.

Many organizations have coding standards that require code to compile without warning messages. There are two ways to eliminate Visual C++'s scanf warnings use scanf_s instead of scanf or disable these warnings. For the input statements we've used so far, Visual C++ users can simply replace scanf with scanf_s. You can disable the warning messages in Visual C++ as follows:

- **1.** Type *Alt F7* to display the **Property Pages** dialog for your project.
- 2. In the left column, expand Configuration Properties > C/C++ and select Preprocessor.
- 3. In the right column, at the end of the value for Preprocessor Definitions, insert ;_CRT_SECURE_NO_WARNINGS
- **4.** Click **OK** to save the changes.

You'll no longer receive warnings on scanf (or any other functions that Microsoft has deprecated for similar reasons). For industrial-strength coding, disabling the warnings is discouraged. We'll say more about scanf_s and printf_s in a later Secure C Coding Guidelines section.

Self Check

[*(Fill-in-the-Blank)* Adding two integers such that the result is a value that's too large to store in an int variable is known as arithmetic _____ and can cause undefined behavior, possibly leaving a system open to attack.

Answer: overflow.

2 (Fill-in-the-Blank) The platform-specific maximum and minimum values that can be stored in an int variable are represented by the constants INT_MAX and INT_MIN, respectively, which are defined in the header _____. Answer: imits.h>.

Summary

Section 3.1 Introduction

 Before writing a program to solve a particular problem, you must have a thorough understanding of the problem and a carefully planned approach to solving it.

Section 3.2 Algorithms

- The solution to any computing problem involves executing a series of actions in a specific order (p. 86).
- An **algorithm** (p. 86) is a **procedure** (p. 86) for solving a problem in terms of the actions (p. 86) to execute and the order in which these actions should execute.

Section 3.3 Pseudocode

- Pseudocode (p. 87) is an artificial and informal language that helps you develop algorithms.
- Pseudocode is similar to everyday English; it's not an actual computer programming language.
- Pseudocode programs help you "think out" a program.
- Pseudocode consists purely of characters. You may type pseudocode using any text editor.
- Carefully prepared pseudocode can be converted easily to corresponding C programs.
- Pseudocode consists only of actions and decisions.

Section 3.4 Control Structures

- Normally, statements in a program execute one after the other in the order in which they're written. This is called **sequential execution** (p. 88).
- Various C statements enable you to specify that the next statement to execute may be other than the next one in sequence. This is called **transfer of control** (p. 88).
- Structured programming has become almost synonymous with "goto elimination" (p. 88).
- Structured programs are clearer, easier to debug and modify and more likely to be bug-free.
- All programs can be written using sequence, selection and iteration control structures (p. 88).
- Unless directed otherwise, the computer automatically executes C statements in sequence.
- A flowchart (p. 88) is a graphical representation of an algorithm drawn using rectangles, diamonds, rounded rectangles and small circles connected by arrows called flowlines (p. 88).
- The **rectangle** (action) symbol (p. 89) indicates any type of action, including a calculation or an input/output operation.
- Flowlines indicate the order in which the actions are performed.
- When drawing a flowchart that represents a complete algorithm, we use as the first symbol a rounded rectangle containing "Begin" and as the last a rounded rectangle containing "End." When drawing only a portion of an algorithm, we omit the rounded-rectangle symbols in favor of using small circles called **connector symbols**.
- The if single-selection statement selects or ignores a single action (or group of actions).
- The **if...else** double-selection statement (p. 89) selects between two different actions (or groups of actions).
- The switch multiple-selection statement (p. 89) selects among many different actions based on the value of an expression.
- C provides three types of iteration statements (also called repetition statements), namely while, do...while and for.
- Control-statement flowchart segments can be attached to one another with control-statement stacking (p. 89)—connecting the exit point of one to the entry point of the next.
- Control statements also may be nested.
- C uses single-entry/single-exit control statements (p. 89).

Section 3.5 The if Selection Statement

- Selection structures are used to choose among alternative courses of action.
- The diamond (decision) symbol (p. 91) indicates that a decision is to be made.
- The **decision symbol**'s expression typically is a condition that can be *true* or *false*. The decision symbol has two flowlines emerging from it indicating the directions to take when the expression is *true* or *false*.
- A decision can be based on any expression's value—zero is *false* and nonzero is *true*.

Section 3.6 The if...else Selection Statement

- The **conditional operator** (?:, p. 92) is closely related to the if...else statement.
- The conditional operator is C's only **ternary operator**—it takes three operands. The first is a condition. The second is the value for the **conditional expression** (p. 92) if the condition is *true*. The third is the value for the conditional expression if the condition is *false*.
- Nested if...else statements (p. 93) test for multiple cases by placing if...else statements inside if...else statements.
- A set of statements within a pair of braces is called a **compound statement** or a **block** (p. 94).
- A syntax error is caught by the compiler. A logic error has its effect at execution time. A fatal logic error causes a program to fail and terminate prematurely. A nonfatal logic error allows a program to continue executing but to produce incorrect results.

Section 3.7 The while Iteration Statement

• The while iteration statement (p. 96) specifies that an action repeats while a condition is *true*. Eventually, the condition will become *false*. At this point, the iteration terminates, and the first statement after the iteration statement executes.

Section 3.8 Formulating Algorithms Case Study 1: Counter-Controlled Iteration

- Counter-controlled iteration (p. 97) uses a variable called a counter (p. 97) to specify the number of times a set of statements should execute.
- Counter-controlled iteration is often called **definite iteration** (p. 97) because the number of iterations is known before the loop begins executing.
- A total (p. 99) is a variable used to accumulate the sum of a series of values. Variables used to store totals should be initialized to zero.
- A **counter** is a variable used to count. Counter variables typically are initialized to zero or one, depending on their use.
- An uninitialized variable contains a "garbage" value (p. 99)—the value last stored in the memory location reserved for that variable.

Section 3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Iteration

- A sentinel value (p. 99; also called a signal value, a dummy value, or a flag value) is used in a sentinel-controlled loop to indicate the "end of data entry."
- Sentinel-controlled iteration is often called indefinite iteration (p. 99) because the number of iterations is not known before the loop begins executing.
- The sentinel value must be chosen so that it cannot be confused with an acceptable input value.

- In top-down, stepwise refinement (p. 100), the top is a statement that conveys the program's overall function. It's a complete representation of a program. In the refinement process, we divide the top into smaller tasks and list these in execution order.
- The type double (p. 102) represents floating-point numbers with decimal points.
- When two integers are divided, any fractional part of the result is truncated (p. 104).
- To produce a floating-point calculation with integer values, you can **cast** the integers to floating-point numbers. C provides the **unary cast operator** (double) to accomplish this task.
- Cast operators (p. 104) perform explicit conversions.
- C requires the operands in arithmetic expressions to have the same data type. To ensure this, the compiler performs **implicit conversion** (p. 104) on selected operands.
- A cast operator is formed by placing parentheses around a type name. The cast operator is a **unary operator**—it takes only one operand.
- Cast operators group right-to-left and have the same precedence as other unary operators such as unary + and unary -. This precedence is one level higher than that of *, / and %.
- The printf conversion Specification %.2f specifies that a floating-point value will be displayed with two digits to the right of the decimal point. If the %f conversion specification is used (without specifying the precision), the **default precision** (p. 105) is 6.
- When floating-point values are printed with precision, the printed value is **rounded** (p. 105) to the indicated number of decimal positions for display purposes.

Section 3.11 Assignment Operators

- C provides several assignment operators for abbreviating assignment expressions (p. 110).
- The += operator adds the value of the expression on its right to the value of the variable on its left and stores the result in the variable on its left.
- Assignment operators are provided for each of the binary operators +, -, *, / and %.

Section 3.12 Increment and Decrement Operators

- C provides the unary increment operator, ++ (p. 111), and the unary decrement operator, -- (p. 111), for use with integral types.
- If ++ or -- operators are placed before a variable, they're referred to as the **preincrement** or **predecrement operators**, respectively. If ++ or -- operators are placed after a variable, they're referred to as the **postincrement** or **postdecrement operators**, respectively.
- Preincrementing (predecrementing) a variable causes it to be incremented (decremented) by 1, then the new value of the variable is used in the expression in which it appears.
- Postincrementing (postdecrementing) a variable uses the current value of the variable in the expression in which it appears, then the variable value is incremented (decremented) by 1.
- When incrementing or decrementing a variable in a statement by itself, pre- and postincrement have the same effect. When a variable appears in the context of a larger expression, pre- and postincrementing have different effects (and similarly for pre- and postdecrementing).

Section 3.13 Secure C Programming

 Adding integers can result in a value that's too large to store in an int variable. This is known as arithmetic overflow and can cause unpredictable runtime behavior, possibly leaving a system open to attack.

- The maximum and minimum values that can be stored in an int variable are represented by the constants INT_MAX and INT_MIN, respectively, from the header <1 imits.h>.
- It's considered good practice to ensure that arithmetic calculations will not overflow before you perform them. In industrial-strength code, you should perform checks for all calculations that can result in overflow or underflow (p. 114).
- The C11 standard's Annex K introduces more secure versions of printf and scanf called printf_s and scanf_s. Annex K is designated as optional, so not every C compiler vendor will implement it.
- Microsoft implemented its own versions of printf_s and scanf_s before the C11 standard and began issuing warnings for every scanf call. The warnings say that scanf is deprecated—it should no longer be used—and that you should consider using scanf_s instead.
- Many organizations have coding standards that require code to compile without warning messages. There are two ways to eliminate Visual C++'s scanf warnings. You can either start using scanf_s immediately or disable this warning message.

Self-Review Exercises

Jell	-INEVIEW LACICISES
3.1	Fill-In the blanks in each of the following questions.a) A procedure for solving a problem in terms of the actions to execute and the order in which the actions should execute is called a(n)
	b) Specifying the execution order of statements by the computer is called
	c) All programs can be written in terms of three types of control statements:
	d) The selection statement is used to execute one action when a condition is <i>true</i> and another action when that condition is <i>false</i> .
	e) Several statements grouped together in braces ({ and }) are called a(n)
	f) The iteration statement specifies that a statement or group of statements is to be executed repeatedly while some condition remains <i>true</i> .
	 g) Iterating a specific number of times is called iteration. h) When it's not known in advance how many times a set of statements will be repeated, a(n) value can be used to terminate the iteration.
3.2	Write four different C statements that each add 1 to integer variable x.
3.3	Write a single C statement to accomplish each of the following: a) Multiply the variable product by 2 using the *= operator.

- b) Multiply the variable product by 2 using the = and * operators.
- c) Test whether the value of the variable count is greater than 10. If it is, print "Count is greater than 10".
- d) Calculate the remainder after quotient is divided by divisor and assign the result to quotient. Write this statement two different ways.
- e) Print the value 123.4567 with two digits of precision. What value is printed?
- f) Print the floating-point value 3.14159 with three digits to the right of the decimal point. What value is printed?

- **3.4** Write a C statement to accomplish each of the following tasks.
 - a) Define variable x to be of type int and set it to 1.
 - b) Define variable sum to be of type int and set it to 0.
 - c) Add variable x to variable sum and assign the result to variable sum.
 - d) Print "The sum is: " followed by the value of variable sum.
- 3.5 Combine the statements from Exercise 3.4 into a program that calculates the sum of the integers from 1 to 10. Use the while statement to loop through the calculation and increment statements. The loop should terminate when x becomes 11.
- **3.6** Write single C statements to perform each of the following tasks:
 - a) Input integer variable x with scanf. Use the conversion specification %d.
 - b) Input integer variable y with scanf. Use the conversion specification %d.
 - c) Set integer variable i to 1.
 - d) Set integer variable power to 1.
 - e) Multiply integer variable power by x and assign the result to power.
 - f) Increment variable i by 1.
 - g) Test i to see if it's less than or equal to y in the condition of a while statement.
 - h) Output integer variable power with printf.
- **3.7** Write a C program that uses the statements in the preceding exercise to calculate x raised to the y power. The program should have a while iteration control statement.
- **3.8** Identify and correct the errors in each of the following:

```
a) while (c <= 5) {
    product *= c;
    ++c;
b) scanf("%.4f", &value);
c) if (gender == 1) {
    puts("Woman");
    }
    else; {
      puts("Man");
    }
</pre>
```

3.9 What's wrong with the following while iteration statement (assume z has value 100), which is supposed to calculate the sum of the integers from 100 down to 1?

```
while (z >= 0) {
   sum += z;
}
```

Answers to Self-Review Exercises

3.1 a) Algorithm. b) Program control. c) Sequence, selection, iteration. d) if...else. e) Compound statement or block. f) while. g) Counter-controlled or definite. h) Sentinel.

Answers to Self-Review Exercises 121 **3.2** See the answer below: x = x + 1;x += 1;++X; X++;**3.3** See the answers below: a) product *= 2; b) product = product * 2; c) if (count > 10) { puts("Count is greater than 10."); d) quotient %= divisor; quotient = quotient % divisor; e) printf("%.2f", 123.4567); 123.46 is displayed. f) printf("%.3f\n", 3.14159); 3.142 is displayed. **3.4** See the answers below: a) int x = 1; b) int sum = 0; c) sum += x; or sum = sum + x; d) printf("The sum is: %d\n", sum); 3.5 See below. I // Calculate the sum of the integers from 1 to 10 #include <stdio.h> 3 4 int main(void) { 5 int x = 1; // set x6 int sum = 0; // set sum 7 8 while $(x \le 10)$ { // loop while x is less than or equal to 10 9 sum += x; // add x to sum++x; // increment x 10 11 } // end while 12 printf("The sum is: %d\n", sum); // display sum 13

```
3.6 See the answers below:
```

14 } // end main function

```
a) scanf("%d", &x);
b) scanf("%d", &y);
c) i = 1;
d) power = 1;
```

e) power *= x;

f) ++i;

```
g) while (i <= y)
h) printf("%d", power);</pre>
```

3.7 See below.

```
// raise x to the y power
 #include <stdio.h>
 3
 4 int main(void) {
 5
       printf("%s", "Enter first integer: ");
 6
       int x = 0:
 7
       scanf("%d", &x); // read value for x from user
       printf("%s", "Enter second integer: ");
 8
 9
       int y = 0;
       scanf("%d", &y); // read value for y from user
10
11
12
       int i = 1;
13
       int power = 1; // set power
14
       while (i <= y) { // loop while i is less than or equal to y
15
          power *= x; // multiply power by x
16
17
          ++i; // increment i
       } // end while
18
19
       printf("%d\n", power); // display power
20
    } // end main function
```

- **3.8** See the answers below:
 - a) Error: Missing the closing right brace of the while body. Correction: Add closing right brace after the statement ++c;.
 - b) Error: Precision used in a scanf conversion specification. Correction: Remove .4 from the conversion specification.
 - c) Error: Semicolon after the else part of the if...else statement results in a logic error. The second puts will always execute.

 Correction: Remove the semicolon after else.
- **3.9** The value of the variable z is never changed in the while statement. Therefore, an infinite loop is created. To prevent the infinite loop, z must be decremented so that it eventually becomes 0.

Exercises

3.10 Identify and correct the errors in each of the following. [*Note:* There may be more than one error in each piece of code.]

```
a) if (age >= 65); {
    puts("Age is greater than or equal to 65");
}
else {
    puts("Age is less than 65");
}
```

```
b) int x = 1;
  int total;

while (x <= 10) {
    total += x;
    ++x;
}
c) While (x <= 100)
    total += x;
    ++x;
d) while (y > 0) {
    printf("%d\n", y);
    ++y;
}
```

- **3.11** Fill-In the blanks in each of the following:
 - a) The solution to any problem involves performing a series of actions in a specific _____.
 - b) A synonym for procedure is _____.
 - c) A variable that accumulates the sum of several numbers is a(n) _____.
 - d) A special value used to indicate "end of data entry" is called a(n) ______, a(n) _____ or a(n) _____ value.
 - e) A(n) _____ is a graphical representation of an algorithm.
 - f) In a flowchart, the order in which the steps should be performed is indicated by _____ symbols.
 - g) Rectangle symbols correspond to calculations that are normally performed by _____ statements and input/output operations that are normally performed by calls to the ____ and ____ Standard Library functions.
 - h) The item written inside a decision symbol is called a(n) _____.
- **3.12** What does the following program print?

```
#include <stdio.h>
 1
 2
 3
    int main(void) {
       int x = 1;
 4
 5
       int total = 0;
 6
 7
       while (x <= 10) {
 8
          int y = x * x;
          printf("%d\n", y);
 9
10
          total += y;
П
          ++X;
       } // end while
12
13
14
       printf("Total is %d\n", total);
    } // end main
```

- **3.13** Write a single pseudocode statement that indicates each of the following:
 - a) Display the message "Enter two numbers".
 - b) Assign the sum of variables x, y, and z to variable p.
 - c) Test the following condition in an if...else selection statement: The current value of variable m is greater than twice the current value of variable v.
 - d) Obtain values for variables s, r, and t from the keyboard.
- **3.14** Formulate a pseudocode algorithm for each of the following:
 - a) Obtain two numbers from the keyboard, compute their sum and display the result.
 - b) Obtain two numbers from the keyboard, and determine and display which (if either) of the two numbers is the larger.
 - c) Obtain a series of positive numbers from the keyboard, and determine and display their sum. Assume that the user types the sentinel value -1 to indicate "end of data entry."
- **3.15** State which of the following are *true* and which are *false*. If a statement is *false*, explain why.
 - a) Experience has shown that the most challenging part of solving a problem on a computer is producing a working C program.
 - b) A sentinel value must be a value that cannot be confused with a legitimate data value.
 - c) Flowlines indicate the actions to be performed.
 - d) Conditions written inside decision symbols always contain arithmetic operators (i.e., +, -, *, /, and %).
 - e) In top-down, stepwise refinement, each refinement is a complete representation of the algorithm.

For Exercises 3.16-3.20, perform each of these steps:

- 1. Read the problem statement.
- 2. Formulate the algorithm using pseudocode and top-down, stepwise refinement.
- 3. Write a C program.
- **4.** Test, debug and execute the C program.
- **3.16** (Gas Mileage) Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several tankfuls of gasoline by recording miles driven and gallons used for each tankful. Develop a program that uses scanf to input the miles driven and gallons used for each tankful. The program should calculate and display the miles per gallon obtained for each tankful. After processing all input information, the program should calculate and print the combined miles per gallon obtained for all tankfuls. Here is a sample input/output dialog:

```
Enter the gallons used (-1 to end): 12.8 Enter the miles driven: 287 The miles/gallon for this tank was 22.421875
```

```
Enter the gallons used (-1 to end): 10.3
Enter the miles driven: 200
The miles/gallon for this tank was 19.417475

Enter the gallons used (-1 to end): 5
Enter the miles driven: 120
The miles/gallon for this tank was 24.000000

Enter the gallons used (-1 to end): -1
The overall average miles/gallon was 21.601423
```

- **3.17** (*Credit-Limit Calculator*) Develop a C program that will determine whether a department-store customer has exceeded the credit limit on a charge account. For each customer, the following facts are available:
 - a) Account number
 - b) Balance at the beginning of the month
 - c) Total of all items charged by this customer this month
 - d) Total of all credits applied to this customer's account this month
 - e) Allowed credit limit

The program should use scanf to input each fact, calculate the new balance (= beginning balance + charges - credits), and determine whether the new balance exceeds the customer's credit limit. For those customers whose credit limit is exceeded, the program should display the customer's account number, credit limit, new balance and the message "Credit limit exceeded." Here is a sample input/output dialog:

```
Enter account number (-1 to end): 100
Enter beginning balance: 5394.78
Enter total charges: 1000.00
Enter total credits: 500.00
Enter credit limit: 5500.00
Account:
                  100
Credit limit: 5500.00
Balance:
                  5894.78
Credit Limit Exceeded.
Enter account number (-1 to end): 200 Enter beginning balance: 1000.00
Enter total charges: 123.45
Enter total credits: 321.00
Enter credit limit: 1500.00
Enter account number (-1 to end): 300 Enter beginning balance: 500.00
Enter total charges: 274.73
Enter total credits: 100.00
Enter credit limit: 800.00
Enter account number (-1 \text{ to end}): -1
```

3.18 (Sales-Commission Calculator) One large chemical company pays its salespeople on a commission basis. The salespeople receive \$200 per week plus 9% of their gross sales for that week. For example, a salesperson who sells \$5000 worth of chemicals in a week receives \$200 plus 9% of \$5000, or a total of \$650. Develop a program that will use scanf to input each salesperson's gross sales for last week and calculate

and display that salesperson's earnings. Process one salesperson's figures at a time. Here is a sample input/output dialog:

```
Enter sales in dollars (-1 to end): 5000.00 Salary is: $650.00

Enter sales in dollars (-1 to end): 1234.56 Salary is: $311.11

Enter sales in dollars (-1 to end): -1
```

3.19 (*Interest Calculator*) The simple interest on a loan is calculated by the formula interest = principal * rate * days / 365;

The preceding formula assumes that rate is the annual interest rate, so it divides by 365 (days per year). Develop a program that uses scanf to input principal, rate and days for several loans, and will calculate and display the simple interest for each loan, using the preceding formula. Here is a sample input/output dialog:

```
Enter loan principal (-1 to end): 1000.00
Enter interest rate: .1
Enter term of the loan in days: 365
The interest charge is $100.00

Enter loan principal (-1 to end): 1000.00
Enter interest rate: .08375
Enter term of the loan in days: 224
The interest charge is $51.40

Enter loan principal (-1 to end): -1
```

3.20 (Salary Calculator) Develop a program that will determine the gross pay for each of several employees. The company pays "straight time" for the first 40 hours worked by each employee and pays "time-and-a-half" for all hours worked in excess of 40 hours. You're given a list of the company's employees, the number of hours each worked last week and each employee's hourly rate. Your program should use scanf to input this information for each employee and determine and display the employee's gross pay. Here is a sample input/output dialog:

```
Enter # of hours worked (-1 to end): 39
Enter hourly rate of the worker ($00.00): 10.00
Salary is $390.00

Enter # of hours worked (-1 to end): 40
Enter hourly rate of the worker ($00.00): 10.00
Salary is $400.00

Enter # of hours worked (-1 to end): 41
Enter hourly rate of the worker ($00.00): 10.00
Salary is $415.00

Enter # of hours worked (-1 to end): -1
```

- **3.21** (*Predecrementing vs. Postdecrementing*) Write a program that demonstrates the difference between predecrementing and postdecrementing using the decrement operator --.
- **3.22** (*Printing Numbers from a Loop*) Write a program that utilizes looping to print the numbers from 1 to 10 side by side on the same line with three spaces between numbers.
- **3.23** (*Find the Largest Number*) Finding the largest number (i.e., the maximum of a group of numbers) is used frequently in computer applications. For example, a program that determines the winner of a sales contest would input the number of units sold by each salesperson. The salesperson who sells the most units wins the contest. Write a pseudocode program and then a program that uses scanf to input a series of 10 non-negative numbers and determines and prints the largest of the numbers. Your program should use three variables:
 - a) counter—A counter to count to 10 (i.e., to keep track of how many numbers have been input and to determine when all 10 numbers have been processed).
 - b) number—The current number input to the program.
 - c) largest—The largest number found so far.
- **3.24** (*Tabular Output*) Write a program that uses looping to print the following table of values. Use the tab escape sequence, \t, in the printf statement to separate the columns with tabs.

N	10*N	100*N	1000*N
1 2	10 20	100 200	1000 2000
3	30	300	3000
4 5	40 50	400 500	4000 5000
6 7	60 70	600 700	6000 7000
8	80	800	8000
9 10	90 100	900 1000	9000 10000
_,	_,,		

3.25 (*Tabular Output*) Write a program that utilizes looping to produce the following table of values:

```
Α
           A+2
                      A+4
                                 A+6
3
6
           8
                      10
                                 12
9
12
                                 15
           11
                      13
           14
                      16
                                 18
15
```

3.26 (Find the Two Largest Numbers) Using an approach similar to Exercise 3.23, find the two largest values of the 10 numbers. You may input each number only once.

- **3.27** (*Validating User Input*) Modify the program in Figure 3.6 to validate its inputs. For each input, if the value is other than 1 or 2, keep looping until the user enters a correct value.
- **3.28** What does the following program print?

```
#include <stdio.h>
2
3
   int main(void) {
4
       int count = 1; // initialize count
5
       while (count \leftarrow 10) { // loop 10 times
6
7
          // output line of text
          puts((count % 2) ? "****" : "+++++++");
8
9
          ++count; // increment count
10
       } // end while
   } // end function main
```

3.29 What does the following program print?

```
#include <stdio.h>
1
2
3
   int main(void) {
       int row = 10; // initialize row
4
5
6
       while (row >= 1) { // loop until row < 1
          int column = 1; // set column to 1 as iteration begins
7
8
9
          while (column <= 10) { // loop 10 times
             printf("%s", (row % 2) ? "<": ">"); // output
10
             ++column; // increment column
11
          } // end inner while
12
13
          --row; // decrement row
14
          puts(""); // begin new output line
15
       } // end outer while
   } // end function main
```

3.30 (Dangling-Else Problem) Determine the output for each of the following when x is 9 and y is 11, and when x is 11 and y is 9. The compiler ignores the indentation in a C program. Also, the compiler always associates an else with the previous if unless told to do otherwise by the placement of braces {}. On first glance, you may not be sure which if an else matches, so this is referred to as the "dangling-else" problem. We eliminated the indentation from the following code to make the problem more challenging. [Hint: Apply indentation conventions you have learned.]

```
a) if (x < 10)
  if (y > 10)
  puts("****");
  else
  puts("####");
  puts("$$$$");
```

```
b) if (x < 10) {
   if (y > 10)
   puts("*****");
   }
   else {
   puts("####");
   puts("$$$$$");
}
```

3.31 (Another Dangling-Else Problem) Modify the following code to produce the output shown. Use proper indentation techniques. You may not make any changes other than inserting braces. The compiler ignores the indentation in a program. We eliminated the indentation from the following code to make the problem more challenging. [Note: It's possible that no modification is necessary.]

```
if (y == 8)
if (x == 5)
puts("@@@@@");
else
puts("####");
puts("$$$$");
puts("&&&&");
```

a) Assuming x = 5 and y = 8, the following output is produced.

```
@@@@@
$$$$$
&&&&&
```

b) Assuming x = 5 and y = 8, the following output is produced.

```
ଉଉଉଉ
```

c) Assuming x = 5 and y = 8, the following output is produced.

```
@@@@@
&&&&&
```

d) Assuming x = 5 and y = 7, the following output is produced.

```
#####
$$$$$
&&&&&
```

3.32 (Square of Asterisks) Write a program that reads in the side of a square and then prints that square out of asterisks. Your program should work for squares of all side sizes between 1 and 20. For example, if your program reads a size of 4, it should print

```
****

***

***

***
```

3.33 (Hollow Square of Asterisks) Modify the program you wrote in the preceding exercise so that it prints a hollow square. For example, if your program reads a size of 5, it should print

```
*****

* *

* *

* *

* *
```

- **3.34** (*Palindrome Tester*) A palindrome is a number or a text phrase that reads the same backward as forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a program that reads in a five-digit integer and determines whether or not it's a palindrome. [*Hint:* Use the division and remainder operators to separate the number into its individual digits.]
- 3.35 (Printing the Decimal Equivalent of a Binary Number) Input a binary integer (5 digits or fewer) containing only 0s and 1s and print its decimal equivalent. [Hint: Use the remainder and division operators to pick off the "binary" number's digits one at a time from right-to-left. Just as in the decimal number system, in which the rightmost digit has a positional value of 1, and the next digit left has a positional value of 10, then 100, then 1000, and so on, in the binary number system the rightmost digit has a positional value of 1, the next digit left has a positional value of 2, then 4, then 8, and so on. Thus the decimal number 234 can be interpreted as 4*1+3*10+2*100. The decimal equivalent of binary 1101 is 1*1+0*2+1*4+1*8 or 1+0+4+8 or 13.]
- **3.36** (*How Fast Is Your Computer?*) How can you determine how fast your own computer operates? Write a program with a while loop that counts from 1 to 1,000,000,000, incrementing by 1 during each iteration of the loop. Every time the count reaches a multiple of 100,000,000, print that number on the screen. Use your watch to time how long each 100 million iterations of the loop takes. [*Hint:* Use the remainder operator to recognize each time the counter reaches a multiple of 100,000,000.]
- **3.37** (*Detecting Multiples of 10*) Write a program that prints 100 asterisks, one at a time. After every tenth asterisk, print a newline character. [*Hint:* Count from 1 to 100. Use the % operator to recognize each time the counter reaches a multiple of 10.]
- **3.38** (*Counting 7s*) Write a program that reads an integer (5 digits or fewer) and determines and prints how many digits in the integer are 7s.
- **3.39** (*Checkerboard Pattern of Asterisks*) Write a program that displays the following checkerboard pattern:

Your program must use only three output statements, one of each of the following forms:

```
printf("%s", "* ");
printf("%s", " ");
puts(""); // outputs a newline
```

- **3.40** (Multiples of 2 with an Infinite Loop) Write a program that keeps printing the multiples of the integer 2, namely 2, 4, 8, 16, 32, 64, and so on. Your loop should not terminate (i.e., you should create an infinite loop). What happens when you run this program?
- **3.41** (*Diameter, Circumference and Area of a Circle*) Write a program that reads the radius of a circle (as a double value) and computes and prints the diameter, the circumference and the area. Use the value 3.14159 for π .
- **3.42** What's wrong with the following statement? Rewrite it to accomplish what the programmer was probably trying to do.

```
printf("%d", ++(x + y));
```

- **3.43** (*Sides of a Triangle*) Write a program that reads three nonzero integer values and determines and prints whether they could represent the sides of a triangle.
- **3.44** (Sides of a Right Triangle) Write a program that reads three nonzero integers and determines and prints whether they could be the sides of a right triangle.
- **3.45** *(Factorial)* The factorial of a non-negative integer *n* is written *n*! (pronounced "*n* factorial") and is defined as follows:

 $n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$ (for values of n greater than or equal to 1) and

$$n! = 1$$
 (for $n = 0$).

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is 120.

- a) Write a program that reads a non-negative integer and computes and prints its factorial.
- b) Write a program that estimates the value of the mathematical constant *e* by using the formula:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

c) Write a program that computes the value of e^x by using the formula

$$e^{x} = 1 + \frac{x}{1!} + \frac{x^{2}}{2!} + \frac{x^{3}}{3!} + \dots$$

3.46 (World Population Growth) World population has grown considerably over the centuries. Continued growth could eventually challenge the limits of breathable air, drinkable water, arable land and other limited resources. There's evidence that growth has been slowing in recent years, and that world population could peak some time this century, then start to decline.

For this exercise, research world population growth issues. This is a controversial topic, so be sure to investigate various viewpoints. Get estimates for the current world population and its growth rate. Write a program that calculates world population growth each year for the next 100 years, using the simplifying assumption that the current growth rate will stay constant. Print the results in a table. The first column should display the year from 1 to 100. The second column should display the anticipated world population at the end of that year. The third column should display the numerical increase in the world population that would occur that year. Using your results, determine the years in which the population would become double and eventually quadruple what it is today.

3.47 (Enforcing Privacy with Cryptography) The explosive growth of Internet communications and data storage on Internet-connected computers has greatly increased privacy concerns. The field of cryptography is concerned with coding data to make it difficult (and hopefully—with the most advanced schemes—impossible) for unauthorized users to read. In this exercise, you'll investigate a simple scheme for encrypting and decrypting data. A company that wants to send data over the Internet has asked you to write a program that will encrypt it so that it may be transmitted more securely. All the data is transmitted as four-digit integers. Your application should read a four-digit integer entered by the user and encrypt it as follows: Replace each digit with the result of adding 7 to the digit and getting the remainder after dividing the new value by 10. Then swap the first digit with the third, and swap the second digit with the fourth. Then print the encrypted integer. Write a separate application that inputs an encrypted four-digit integer and *decrypts* it (by reversing the encryption scheme) to form the original number. [Optional reading project: In industrial-strength applications, you'll want to use much stronger encryption techniques than presented in this exercise. Research "public-key cryptography" in general and the PGP (Pretty Good Privacy) specific public-key scheme. You may also want to investigate the RSA scheme, which is widely used in industrial-strength applications.

Program Control



4

Objectives

In this chapter, you'll:

- Learn the essentials of countercontrolled iteration.
- Use the for and do...while iteration statements to execute statements repeatedly.
- Understand multiple selection using the switch selection statement.
- Use the break and continue statements to alter the flow of control.
- Use logical operators to form complex conditions in control statements.
- Avoid the consequences of confusing the equality and assignment operators.

- 4.1 Introduction
- **4.2** Iteration Essentials
- 4.3 Counter-Controlled Iteration
- **4.4 for** Iteration Statement
- **4.5** Examples Using the **for** Statement
- **4.6 switch** Multiple-Selection Statement

- **4.7** do...while Iteration Statement
- 4.8 break and continue Statements
- 4.9 Logical Operators
- **4.10** Confusing Equality (==) and Assignment (=) Operators
- **4.11** Structured-Programming Summary
- **4.12** Secure C Programming

Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises

4.1 Introduction

You should now be comfortable with reading and writing simple C programs. Next, we consider iteration in more detail and introduce C's for and do...while iteration statements. We also introduce:

- the switch multiple-selection statement,
- the break statement for exiting immediately from certain control statements, and
- the continue statement for skipping the remainder of an iteration statement's body then proceeding with the next iteration of the loop.

We also discuss logical operators used for combining conditions and summarizes the principles of structured programming as presented here and in Chapter 3.

4.2 Iteration Essentials

Most programs involve iteration (or looping). A loop is a group of instructions the computer repeatedly executes while some loop-continuation condition remains true. We've discussed two means of iteration:

- 1. Counter-controlled iteration.
- 2. Sentinel-controlled iteration.

You saw in Chapter 3 that counter-controlled iteration uses a control variable to count the number of iterations for a group of instructions to perform. When the control variable's value indicates that the correct number of iterations has been completed, the loop terminates, and execution continues with the statement after the iteration statement.

You saw in Chapter 3 that we use sentinel values to control iteration if the precise number of iterations isn't known in advance, and the loop includes statements that obtain data each time the loop is performed. A sentinel value indicates "end of data." The sentinel is entered after all regular data items have been supplied to the program. Sentinels must be distinct from regular data items.

Self Check

(*Fill-In*) A loop is a group of instructions the computer repeatedly executes while a _____ condition remains *true*.

Answer: loop-continuation.

- **2** (*Multiple Choice*) Which of the following statements is *false?*
 - a) Sentinel values are used to control iteration when the precise number of iterations isn't known in advance, and the loop includes statements that obtain data each time the loop is performed.
 - b) The sentinel value indicates "end of data."
 - c) The sentinel is entered after all regular data items have been supplied.
 - d) The sentinel must match a regular data item.

Answer: d) is false. Actually, the sentinel must be distinct from regular data items.

4.3 Counter-Controlled Iteration

Counter-controlled iteration requires:

- the name of a control variable,
- the initial value of the control variable,
- the increment (or decrement) by which the control variable is modified in each iteration, and
- the loop-continuation condition that tests for the **final value** of the control variable to determine whether looping should continue.

Consider Fig. 4.1, which displays the numbers from 1 through 5. The definition

```
int counter = 1; // initialization
```

names the control variable (counter), defines it as an integer, reserves memory space for it, and sets its initial value to 1.

```
// fig04_01.c
2
    // Counter-controlled iteration.
3
    #include <stdio.h>
5
    int main(void) {
6
        int counter = 1; // initialization
7
8
       while (counter <= 5) { // iteration condition
           printf("%d ", counter);
++counter; // increment
9
10
11
        }
12
        puts("");
    }
14
```

Fig. 4.1 | Counter-controlled iteration.

1 2 3 4 5

The statement

```
++counter; // increment
```

increments counter by 1 at the end of each loop iteration. The while's condition

```
counter <= 5
```

tests whether the value of the control variable is less than or equal to 5 (the last value for which the condition is true). This while terminates when the control variable exceeds 5 (i.e., counter becomes 6).

Use Integer Counters

Floating-point values may be approximate, so controlling counting loops with floating-point variables may result in imprecise counter values and inaccurate termination tests. For this reason, you should always control counting loops with integer values.

Self Check

- **I** (*Multiple Choice*) Which of the following a), b) or c) is required by counter-controlled iteration?
 - a) The name and initial value of a control variable (or loop counter).
 - b) The increment (or decrement) by which the control variable is modified each time through the loop.
 - c) The loop-continuation condition that tests for the final value of the control variable to determine whether looping should continue.
 - d) All of the above are required by counter-controlled iteration.

Answer: d.

- **2** (Multiple Choice) Based on this section's program, which of the following statements a), b) or c) is false?
 - a) Control variable counter increments by 1 during each iteration of the loop.
 - b) The loop terminates when counter is 5.
 - c) The while's body executes even when the control variable is 5.
 - d) All of the above statements are true.

Answer: b) is *false*. Actually, the loop terminates when the control variable becomes 6.

4.4 for Iteration Statement

The for iteration statement (lines 8–10 of Fig. 4.2) handles all the details of counter-controlled iteration. For readability, try to fit the for statement's header (line 8) on one line. The for statement executes as follows:

- When it begins executing, the for statement defines the control variable counter and initializes it to 1.
- Next, it tests its loop-continuation condition counter <= 5. The initial value of counter is 1, so the condition is *true*, and the for statement executes its printf statement (line 9) to display counter's value, namely 1.

- Next, the for statement increments the control variable counter using the expression ++counter, then re-tests the loop-continuation condition. The control variable is now equal to 2, so the condition is still *true*, and the for statement executes its printf statement again.
- This process continues until the control variable counter becomes 6. At this point, the loop-continuation condition is *false* and iteration terminates.

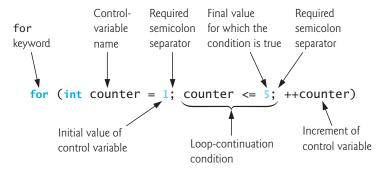
The program continues executing with the first statement after the for (line 12).

```
// fig04_02.c
   // Counter-controlled iteration with the for statement.
2
3
   #include <stdio.h>
5
   int main(void) {
       // initialization, iteration condition, and increment
6
7
       // are all included in the for statement header.
8
       for (int counter = 1; counter <= 5; ++counter) {</pre>
9
          printf("%d ", counter);
10
11
12
       puts(""); // outputs a newline
    }
13
   2 3 4 5
```

Fig. 4.2 | Counter-controlled iteration with the for statement.

for Statement Header Components

The following diagram takes a closer look at Fig. 4.2's for statement, which specifies each of the items needed for counter-controlled iteration. If there's more than one statement in the for's body, braces are required. As with the other control statements, always place a for statement's body in braces, even if it has only one statement.



Control Variables Defined in a for Header Exist Only Until the Loop Terminates

When you define the control variable in the for header before the first semicolon (;), as in line 8 of Fig. 4.2:

```
for (int counter = 1; counter <= 5; ++counter) {</pre>
```

the control variable exists only until the loop terminates. So, attempting to access the ERR (x) control variable after the for statement's closing right brace (3) is a compilation error.

Off-By-One Errors

If we had written the loop-continuation condition counter <= 5 as counter < 5, then the loop would be executed only four times. This is a common logic error called an ERR (x) off-by-one error. Using a control variable's final value in a while or for statement condition and using the <= relational operator can help avoid off-by-one errors. To print the values 1 to 5, for example, the loop-continuation condition should be counter <= 5 rather than counter < 6.

General Format of a for Statement

The general format of the for statement is

```
for (initialization; loopContinuationCondition; increment) {
   statement
}
```

where

- *initialization* names the loop's control variable and provides its initial value,
- loop Continuation Condition determines whether the loop should continue executing, and
- increment modifies the control variable's value after executing the statement so that the loop-continuation condition eventually becomes *false*.

The two semicolons in the for header are required. If the loop-continuation condition is initially *false*, the program does not execute the for statement's body. Instead, execution proceeds with the statement following the for.



Infinite loops occur when the loop-continuation condition never becomes *false*. To prevent infinite loops, ensure that you do not place a semicolon immediately after a while statement's header. In a counter-controlled loop, ensure that you increment (or decrement) the control variable so the loop-continuation condition eventually becomes *false*. In a sentinel-controlled loop, ensure that the sentinel value is eventually input.

Expressions in the for Statement's Header Are Optional

All three expressions in a for header are optional. If you omit the loop Continuation-Condition, the condition is always true, thus creating an infinite loop. You might omit the *initialization* expression if the program initializes the control variable before the loop. You might omit the increment expression if the program calculates the increment in the loop's body or if no increment is needed.

Increment Expression Acts Like a Standalone Statement

The for statement's *increment* acts like a standalone C statement at the end of the for's body. So, the following are all equivalent in a for statement's increment expression:

```
counter = counter + 1
counter += 1
++counter
counter++
```

The increment in a for statement's *increment* expression may be negative, in which case it's a *decrement*, and the loop counts *downward*.

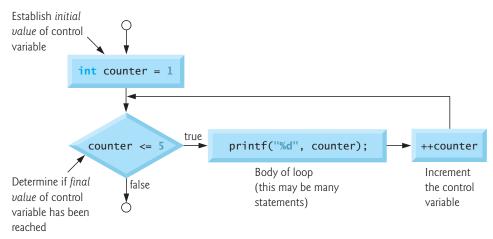
Using a for Statement's Control Variable in the Statement's Body

Programs frequently display the control-variable value or use it in calculations in the loop body, but this use is not required. The control variable is commonly used to control iteration without being mentioned in the for statement's body. Although the control variable's value can be changed in a for loop's body, avoid doing so, because this practice can lead to subtle errors. It's best not to change it.



for Statement Flowchart

Below is the flowchart for the for statement in Fig. 4.2:



This flowchart makes it clear that the initialization occurs once, and the increment occurs *after* the body statement each time it's performed.

Self Check

- I (*TruelFalse*) When you define the control variable in the for header before the first semicolon (;), the control variable exists only until the loop terminates. **Answer:** True.
- **2** (*Multiple Choice*) Which of the following statements a), b) or c) is *true*?
 - a) The for statement header specifies each of the items needed for counter-controlled iteration with a control variable.
 - b) If there's more than one statement in a for's body, braces are required.
 - c) You should always place a control statement's body in braces, even if it has only one statement.
 - d) All of the above statements are true.

Answer: d.

4.5 Examples Using the for Statement

The following examples show ways to vary the control variable in a for statement.

1. Vary the control variable from 1 to 100 in increments of 1.

```
for (int i = 1; i \le 100; ++i)
```

2. Vary the control variable from 100 to 1 in increments of -1 (i.e., *decrements* of 1).

```
for (int i = 100; i >= 1; --i)
```

3. Vary the control variable from 7 to 77 in increments of 7.

```
for (int i = 7; i <= 77; i += 7)
```

4. Vary the control variable from 20 to 2 in increments of -2.

```
for (int i = 20; i >= 2; i -= 2)
```

5. Vary the control variable over the values 2, 5, 8, 11, 14 and 17.

```
for (int j = 2; j <= 17; j += 3)
```

6. Vary the control variable over the following sequence of values: 44, 33, 22, 11, 0.

```
for (int j = 44; j >= 0; j -= 11)
```

Application: Summing the Even Integers from 2 to 100

Figure 4.3 uses the for statement to sum the even integers from 2 to 100. Each loop iteration (lines 8–10) adds the control variable number's current value to the sum.

```
I // fig04_03.c
   // Summation with for.
3
   #include <stdio.h>
4
5
   int main(void) {
6
       int sum = 0; // initialize sum
7
8
       for (int number = 2; number <= 100; number += 2) {
9
          sum += number; // add number to sum
10
11
       printf("Sum is %d\n", sum);
12
13
   }
```

Sum is 2550

Fig. 4.3 | Summation with for.

Application: Compound-Interest CalculationsThe next example computes compound interest using the for statement. Consider

formula for determining these amounts:

the following problem statement:

A person invests \$1000.00 in a savings account yielding 5% interest. Assuming all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following

```
a = p(1 + r)^n
where

p is the original amount invested (i.e., the principal, $1000.00 here),
r is the annual interest rate (for example, .05 for 5%),
n is the number of years, which is 10 here, and
a is the amount on deposit at the end of the n<sup>th</sup> year.
```

The solution (Fig. 4.4) uses a counter-controlled loop to perform the same calculation for each of the 10 years the money remains on deposit. The for statement executes its body 10 times, varying a control variable from 1 to 10 in increments of 1. C does *not* include an exponentiation operator, so we use the Standard Library function pow (line 17) for this. The call pow(x, y) calculates x raised to the yth power. The function takes two arguments of data type double. When it completes its calculation, pow returns (that is, gives back) a double value, which we then multiply by principal (line 17).

```
// fig04_04.c
 1
   // Calculating compound interest.
   #include <stdio.h>
 3
 4
   #include <math.h>
 5
 6
    int main(void) {
 7
       double principal = 1000.0; // starting principal
 8
       double rate = 0.05; // annual interest rate
 9
10
       // output table column heads
       printf("%4s%21s\n", "Year", "Amount on deposit");
П
12
       // calculate amount on deposit for each of ten years
13
14
       for (int year = 1; year \leftarrow 10; ++year) {
15
16
           // calculate new amount for specified year
          double amount = principal * pow(1.0 + rate, year);
17
18
19
          // output one table row
20
          printf("%4d%21.2f\n", year, amount);
21
       }
    }
22
Year
        Amount on deposit
   1
                   1050.00
   2
                   1102.50
   3
                   1157.63
   4
                   1215.51
   5
                   1276.28
   6
                   1340.10
   7
                   1407.10
   8
                   1477.46
   9
                   1551.33
```

Fig. 4.4 | Calculating compound interest.

1628.89

10

You must include <math.h> (line 4) to use pow and C's other math functions.¹ If you did not include the header, this program would malfunction, as the linker would be unable to find the pow function. Function pow requires two double arguments, but variable year is an integer. The math.h file includes information that tells the compiler to convert the year value to a temporary double representation before calling pow. This information is contained in pow's function prototype. We explain function prototypes in Chapter 5, where we also summarize many other math library functions.

Formatting Numeric Output

This program used the conversion specification %21.2f to print variable amount's value. The 21 in the conversion specification denotes the field width in which the value will be printed. A field width of 21 specifies that the value printed will use 21 character positions. As you learned in Chapter 3, the .2 specifies the precision (i.e., the number of decimal positions). If the number of characters displayed is less than the field width, then the value will be right-aligned with leading spaces. This is particularly useful for aligning the decimal points of floating-point values vertically. To left-align a value in a field, place a - (minus sign) between the % and the field width. We'll discuss the powerful formatting capabilities of printf and scanf in detail in Chapter 9.

Floating-Point Number Precision and Memory Requirements

Variables of type float typically require four bytes of memory with approximately seven significant digits. Variables of type double typically require eight bytes of memory with approximately 15 significant digits—about double the precision of floats. Most programmers use type double. C treats floating-point values such as 3.14159 as type double by default. Such values in the source code are known as floating-point literals.

C also has type long double. Such variables typically are stored in 12 or 16 bytes of memory. The C standard states the minimum sizes of each floating-point type and indicates that type double provides at least as much precision as float and that type long double provides at least as much precision as double. For a list of C's fundamental numeric types and their typical ranges, see

https://en.cppreference.com/w/c/language/arithmetic_types

Floating-Point Numbers Are Approximations

In conventional arithmetic, floating-point numbers often arise as a result of division—when we divide 10 by 3, the result is the infinitely repeating sequence 3.3333333.... with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so the stored floating-point value can be only an approximation. So, C's floating-point types suffer from what is referred to as representational error. Assuming that floating-point numbers are represented exactly (e.g., using them in comparisons for equality) can lead to incorrect results.

Floating-point numbers have numerous applications, especially for measured values. For example, when we speak of a "normal" body temperature of 98.6 degrees

^{1.} For the gcc compiler, you must include the -1m option (e.g., gcc -1m fig04_04.c) when compiling Fig. 4.4. This links the math library to the program.

Fahrenheit, we do not need to be precise to a large number of digits. When we read the temperature on a thermometer as 98.6, it actually might be 98.5999473210643. Calling this number 98.6 is fine for most applications involving body temperatures.

A Warning about Displaying Rounded Values

We declared variables amount, principal and rate to be of type double in this example. We're dealing with fractional parts of dollars and thus need a type that allows decimal points in its values. Unfortunately, floating-point numbers can cause trouble. Here's a simple explanation of what can go wrong when using floating-point numbers to represent dollar amounts displayed with two digits to the right of the decimal point. Two calculated dollar amounts stored in the machine could be 14.234 (rounded to 14.23 for display purposes) and 18.673 (rounded to 18.67 for display purposes). When these amounts are added, they produce the internal sum 32.907, which would typically be rounded to 32.91 for display purposes. Thus, your output could appear as

but a person adding the individual numbers as displayed would expect the sum to be 32.90. You've been warned!

Common Dollar Amounts Can Have Floating-Point Representational Errors

Even simple dollar amounts, such as those you might see on a grocery or restaurant bill, can have representational errors when they're stored as doubles. To see this, we created a simple program with the declaration

```
double d = 123.02;
```

then displayed d's value with many digits of precision to the right of the decimal point. The output showed 123.02 as 123.0199999..., which is another example of a representational error. Though some dollar amounts can be represented precisely as double, many cannot. This is a common problem in many programming languages.

Self Check

I (Fill-In) To _____ a value in a field, place a - (minus sign) between the % and the field width.

Answer: left-align.

- **2** (*Multiple Choice*) Which of the following for statement headers is incorrect?
 - a) Vary the control variable from 1 to 100 in increments of 1.

```
for (int i = 1; i \le 100; ++i)
```

b) Vary the control variable from 7 to 77 in increments of 7.

for (int
$$i = 7$$
; $i <= 77$; $i += 7$)

c) Vary the control variable over the following sequence: 2, 5, 8, 11, 15, 17.

```
for (int j = 2; j <= 17; j += 3)
```

d) Vary the control variable from 20 to 2 in increments of -2.

```
for (int i = 20; i >= 2; i -= 2)
```

Answer: c) is incorrect. The for statement actually generates the sequence 2, 5, 8, 11, 14, 17. It does not generate the value 15 in the original series.

4.6 switch Multiple-Selection Statement

In Chapter 3, we discussed the if single-selection and the if...else double-selection statements. Occasionally, an algorithm will contain a series of decisions that test a variable or expression separately for each of the integer values it may assume, then perform different actions. This is called multiple selection. C provides the switch multiple-selection statement to handle such decision making.

The switch statement consists of a series of case labels, an optional default case and statements to execute for each case. Figure 4.5 uses switch to count the number of each different letter grade students earned on an exam.

```
I // fig04_05.c
 2 // Counting letter grades with switch.
 3 #include <stdio.h>
 4
 5 int main(void) {
 6
       int aCount = 0;
 7
       int bCount = 0;
       int cCount = 0;
 8
 9
       int dCount = 0;
10
       int fCount = 0;
11
       puts("Enter the letter grades.");
12
13
       puts("Enter the EOF character to end input.");
       int grade = 0; // one grade
14
15
16
       // loop until user types end-of-file key sequence
       while ((grade = getchar()) != EOF) {
17
18
19
          // determine which grade was input
          switch (grade) { // switch nested in while
20
             case 'A': // grade was uppercase A
21
22
             case 'a': // or lowercase a
23
                ++aCount;
24
                break; // necessary to exit switch
25
             case 'B': // grade was uppercase B
             case 'b': // or lowercase b
26
27
                ++bCount;
                break;
28
             case 'C': // grade was uppercase C
29
             case 'c': // or lowercase c
30
31
                ++cCount;
32
                break:
```

Fig. 4.5 | Counting letter grades with switch. (Part 1 of 2.)

```
case 'd': // or lowercase d
34
35
                  ++dCount;
36
                  break;
              case 'F': // grade was uppercase F
37
              case 'f': // or lowercase f
38
39
                  ++fCount;
40
                  break;
              case '\n': // ignore newlines,
41
              case '\t': // tabs,
42
              case ' ': // and spaces in input
43
44
                  break;
45
              default: // catch all other characters
                  printf("%s", "Incorrect letter grade entered.");
46
47
                  puts(" Enter a new grade.");
                  break; // optional; will exit switch anyway
48
49
           } // end switch
        } // end while
50
51
52
        // output summary of results
53
        puts("\nTotals for each letter grade are:");
       printf("A: %d\n", aCount);
printf("B: %d\n", bCount);
54
55
        printf("C: %d\n", cCount);
56
       printf("D: %d\n", dCount);
printf("F: %d\n", fCount);
57
58
59 }
Enter the letter grades.
Enter the EOF character to end input.
b
C
C
Α
d
f
C
Incorrect letter grade entered. Enter a new grade.
D
Α
b

    Not all systems display a representation of the EOF character

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

case 'D': // grade was uppercase D

Fig. 4.5 | Counting letter grades with **switch**. (Part 2 of 2.)

33

Reading Character Input

In the program, the user enters students' letter grades. In the while header (line 17),

```
while ((grade = getchar()) != EOF)
```

the parenthesized assignment (grade = getchar()) executes first. The getchar function (from <stdio.h>) reads one character from the keyboard and stores that character in the integer variable grade. Characters are normally stored in **char** variables. However, C can store characters in variables of any integer type because characters are usually represented as one-byte integers in the computer. Function getchar returns as an int the character that the user entered. We can treat a character as either an integer or a character, depending on its use. For example, the statement

```
printf("The character (%c) has the value %d.\n", 'a', 'a');
```

uses the conversion specifications %c and %d to print the character 'a' and its integer value. The result is

```
The character (a) has the value 97.
```

Characters can be read with scanf by using the conversion specification %c. The integer 97 is the numerical representation of the character 'a' in the computer. Many computers today use the Unicode[®] character set. Appendix B contains the ASCII (American Standard Code for Information Interchange) character set and its numeric values. ASCII is a subset of Unicode.

Assignments Have Values

Assignments as a whole actually have a value. The value of the assignment expression grade = getchar() is the character that's returned by getchar and assigned to the variable grade. The fact that assignments have values can be useful for setting several variables to the same value. For example,

```
a = b = c = 0;
```

first evaluates the assignment c = 0 (because the = operator groups from right to left). The variable b is then assigned the value of the assignment c = 0 (which is 0). Then, the variable a is assigned the value of the assignment b = (c = 0) (which is also 0).

The value of the assignment grade = getchar() is compared with the value of EOF (a symbol whose acronym stands for "end of file"). We use EOF (which normally has the value -1) as the sentinel value. The user types a system-dependent keystroke combination to mean "end of file"—i.e., "I have no more data to enter." EOF is a symbolic integer constant defined in the <stdio.h> header (we'll see in Chapter 6 how symbolic constants are defined). If the value assigned to grade is equal to EOF, the program terminates.

We represent characters in this program as ints because EOF has an integer value (again, normally -1). Testing for the symbolic constant EOF, rather than -1, makes programs more portable. The C standard states that EOF is a negative integral value (but not necessarily -1). Thus, EOF could have different values on different systems.

Entering the EOF Indicator

The keystroke combinations for entering EOF (end of file) are system dependent. On Linux/UNIX/macOS systems, the EOF indicator is entered by typing on a line by itself

$$Ctrl + d$$

This notation means to simultaneously press both the Ctrl key and the d key. On other systems, such as Microsoft Windows, the EOF indicator can be entered by typing

$$Ctrl + z$$

You also need to press *Enter* on Windows.

The user enters grades at the keyboard. When the *Enter* key is pressed, the characters are read by function getchar one at a time. If the character entered is not equal to EOF, the switch statement (lines 20–49) executes.

switch Statement Details

Keyword switch is followed by the variable name grade in parentheses. This is called the **controlling expression**. The switch compares this expression's value with each of the **case labels**. Each case can have one or more actions, but braces are not required around multiple actions in a given case.

Assume the user has entered the letter C as a grade. When the switch compares C to each case, if a match occurs (case 'C':), the statements for that case execute. For the letter C, the switch increments cCount by 1 (line 31), then the break statement (line 32) exits the switch immediately, causing program control to continue with the first statement after the switch statement.

We use a break statement here because the cases in a switch statement would otherwise run together. Without break statements, each time a match occurs, all the remaining cases' statements will execute. (This feature—called fallthrough—is rarely useful, although it's perfect for compactly programming Exercise 4.38—the iterative song "The Twelve Days of Christmas"!) Forgetting a break statement when one is needed in a switch statement is a logic error.



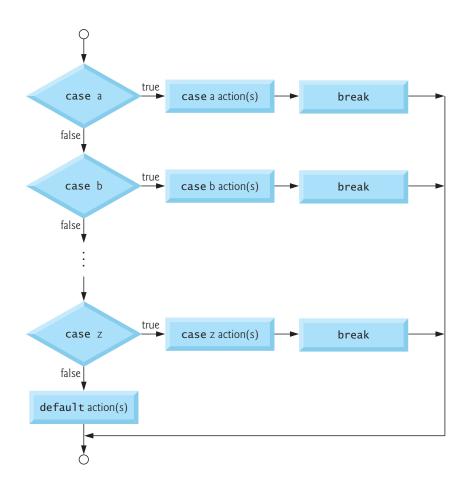
default Case

If no match occurs, the default case executes. In this program, it displays an error message. You should always include a default case; otherwise, values not explicitly tested in a switch will be ignored. The default case helps prevent this by focusing you on the need to process exceptional conditions. Sometimes no default processing is needed.

Although the case clauses and the default case clause in a switch statement can occur in any order, it's common to place the default clause last. When the default clause is last, the break statement isn't required. But many programmers include this break for clarity and symmetry with other cases.

switch Statement Flowchart

The following switch multiple-selection-statement flowchart makes it clear that each case's break statement immediately exits the switch statement.



Ignoring Newline, Tab and Blank Characters in Input

In the switch statement of Fig. 4.5, the lines

```
case '\n': // ignore newlines,
case '\t': // tabs,
case ' ': // and spaces in input
  break;
```

cause the program to skip newline, tab and blank characters. Reading characters one at a time can cause problems. To have the program read the characters, you must send them to the computer by pressing *Enter*. This places the newline character in the input after the character we wish to process.

Often, this newline (and other whitespace characters) must be specifically ignored to make the program work correctly. The preceding cases in our switch statement prevent the error message in the default case from being printed each time a newline, tab or space is encountered in the input. Each input in this example causes two iterations of the loop—the first for a letter grade and the second for '\n'. Listing several case labels with no intervening statements means that the same actions occur for each of the cases.

Constant Integral Expressions

When using the switch statement, remember that each case can test only a constant integral expression. The expression can be any combination of character constants and integer constants that evaluates to a constant integer value. A character constant can be represented as the specific character in single quotes, such as 'A'. Characters must be enclosed within single quotes to be recognized as character constants—characters in double quotes are recognized as strings. Integer constants are simply integer values. In our example, we used character constants.

Notes on Integral Types

Portable languages like C must have flexible data-type sizes. Applications may need integers of various sizes. C provides several data types to represent integers. In addition to int and char, C provides types short int (which can be abbreviated as short) and long int (which can be abbreviated as long). There also are unsigned variations of all the integral types that represent non-negative integer values. In Section 5.14, we'll see that C also provides type long long int (which can be abbreviated as long long).

The C standard specifies the minimum range of values for each integer type. The actual range may be greater, depending on the implementation. For short ints, the minimum range is -32767 to +32767. For most integer calculations, long ints are sufficient. The minimum range of values for long ints is -2147483647 to +2147483647. An int's range is greater than or equal to that of a short int and less than or equal to that of a long int. On many of today's platforms, ints and long ints represent the same range of values. The data type signed char can represent integers in the range -127 to +127 or any of the ASCII character set. See Section 5.2.4.2 of the C standard document for the complete list of signed and unsigned integer-type minimum ranges.

Self Check

- I (*Fill-In*) Occasionally, an algorithm will contain a series of decisions in which a variable or expression is tested separately for each of the constant integral values it may assume, and different actions are taken. This is called _____. Answer: multiple selection.
- **2** (Multiple Choice) Which of the following statements a), b) or c) is false?
 - a) The value of an assignment is the value assigned to the variable on the left of the =.
 - b) The value of the assignment expression grade = getchar() is the character that's returned by getchar and assigned to the variable grade.
 - c) The following statement sets variables a, b and c to 0:

$$0 = a = b = c;$$

d) All of the above statements are true.

Answer: c) is *false*. The correct statement is:

$$a = b = c = 0;$$

4.7 do...while Iteration Statement

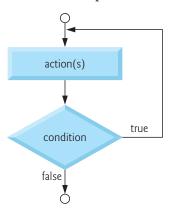
The do...while iteration statement is similar to the while statement. The while statement tests its loop-continuation condition before executing the loop body. The do...while statement tests its loop-continuation condition after executing the loop body, so the loop body always executes at least once. When a do...while terminates, execution continues with the statement after the while clause. Figure 4.6 uses a do...while statement to display the numbers from 1 through 5. We chose to preincrement the control variable counter in the loop-continuation test (line 10).

```
// fig04_06.c
I
   // Using the do...while iteration statement.
2
    #include <stdio.h>
5
    int main(void) {
6
       int counter = 1; // initialize counter
7
8
       do {
          printf("%d ", counter);
9
       } while (++counter <= 5);</pre>
10
    }
11
   2 3 4 5
```

Fig. 4.6 Using the **do**...**while** iteration statement.

do...while Statement Flowchart

The following do...while statement flowchart makes it clear that the loop-continuation condition does not execute until after the loop's action is performed the first time:



Self Check

- (Multiple Choice) Which of the following statements a), b) or c) is false?
 - a) The while statement tests its loop-continuation condition before executing its body, so the loop body will always execute at least once.
 - b) The do...while statement tests its loop-continuation condition after executing its loop body.

- c) When a do...while terminates, execution continues with the statement after the while clause.
- d) All of the above statements are *true*.

Answer: a) is *false*. Actually, if the while statement's loop-continuation test fails upon entering the loop, the loop's body will not execute.

2 (*True/False*) Assuming counter is initialized to 1, the following loop displays the numbers 1 through 10:

```
do {
   printf("%d ", counter);
} while (++counter < 10);</pre>
```

Answer: *False.* This loop displays the numbers 1 through 9. To display the numbers 1 through 10, change the < in the loop-continuation condition to <=.

4.8 break and continue Statements

The break and continue statements are used to alter the flow of control. Section 4.6 showed that a break encountered in a switch statement terminates the switch's execution. This section discusses how to use break in an iteration statement.

break Statement

The break statement, when executed in a while, for, do...while or switch statement, causes an immediate exit from that statement. Program execution continues with the next statement after that while, for, do...while or switch. Common uses of break are to escape early from a loop or skip the remainder of a switch (as in Fig. 4.5). Figure 4.7 demonstrates the break statement (line 12) in a for iteration statement.

```
// fig04_07.c
   // Using the break statement in a for statement.
2
   #include <stdio.h>
5
   int main(void) {
       int x = 1; // declared here so it can be used after loop
6
7
8
       // loop 10 times
9
       for (; x \le 10; ++x) {
          // if x is 5, terminate loop
10
          if (x == 5) {
11
             break; // break loop only if x is 5
12
13
14
15
          printf("%d ", x);
16
17
       printf("\nBroke out of loop at x == %d\n", x);
18
```

Fig. 4.7 Using the break statement in a for statement. (Part 1 of 2.)

```
1 2 3 4
Broke out of loop at x == 5
```

Fig. 4.7 Using the **break** statement in a **for** statement. (Part 2 of 2.)

When the if statement detects that x has become 5, break executes. This terminates the for statement, and the program continues with the printf after the for. The loop fully executes only four times. Recall that when you declare the control variable in a for loop's *initialization* expression, the variable no longer exists after the loop terminates. We declared and initialized x before the loop in this example, so that we could use its final value after the loop terminates. So, the initialization section of the for's header (before the first semicolon) is empty.

continue Statement

The continue statement, when executed in a while, for or do...while statement, skips the remaining statements in that control statement's body and performs the next iteration of the loop. In while and do...while statements, the loop-continuation test is evaluated immediately after the continue statement executes. In the for statement, the increment expression executes, then the loop-continuation test is evaluated. Figure 4.8 uses continue (line 10) in the for statement to skip the printf statement when x is 5 and begin the next iteration of the loop.

```
// fig04_08.c
1
   // Using the continue statement in a for statement.
   #include <stdio.h>
3
4
5
   int main(void) {
6
       // loop 10 times
7
       for (int x = 1; x <= 10; ++x) {
8
          // if x is 5, continue with next iteration of loop
          if (x == 5) {
9
             continue; // skip remaining code in loop body
10
11
12
          printf("%d ", x);
13
       }
14
15
16
       puts("\nUsed continue to skip printing the value 5");
17
    }
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Fig. 4.8 Using the **continue** statement in a **for** statement.

break and continue Notes

Some programmers feel break and continue violate the norms of structured programming, so they do not use them. The effects of these statements can be achieved by

structured programming techniques we'll soon discuss, but the break and continue statements perform faster.

There's a tension between achieving quality software engineering and achieving 🔺 SE the best-performing software—one is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following guidelines: First, make your code simple and correct; then make it fast and small, but only if necessary.

Self Check

- (Multiple Choice) Which of the following statements a), b) or c) is false?
 - a) The break statement terminates a switch statement's execution.
 - b) The break statement, when executed in a while, for or do...while statement, causes an immediate exit from that statement.
 - c) Common uses of break are to escape early from a loop or to skip the remainder of an if...else.
 - d) All of the above statements are *true*.

Answer: c) is false. Actually, break skips the remainder of a switch, not an if...else.

- **2** (*Multiple Choice*) Which of the following statements a), b) or c) is *false*?
 - a) The continue statement, when executed in a while, for or do...while statement, skips the remaining statements in that control statement's body and performs the next iteration of the loop.
 - b) In while and do...while statements, the loop-continuation test is evaluated immediately after the continue statement executes.
 - c) In the for statement, after the continue statement executes, the loop-continuation test is evaluated, then the increment expression executes.
 - d) All of the above statements are *true*.

Answer: c) is false. Actually, in the for statement, after the continue statement executes, the increment expression executes, then the loop-continuation test is evaluated.

4.9 Logical Operators

So far we've used simple conditions, such as counter <= 10, total > 1000, and grade != -1. We've expressed these conditions in terms of the relational operators (>, <, >= and <=) and equality operators (== and !=). Each decision tested precisely one condition. To test multiple conditions in the process of making a decision, we had to perform these tests in separate statements or in nested if or if...else statements. C provides logical operators that may be used to form more complex conditions by combining simple conditions. The logical operators are && (logical AND), || (logical OR) and ! (logical NOT, which is also called logical negation). We'll consider examples of each of these operators.

Logical AND (&&) Operator

Suppose we wish to ensure that two conditions are both true before we choose a certain path of execution. In this case, we can use the logical operator && as follows:

```
if (gender == 1 && age >= 65) {
    ++seniorFemales;
}
```

This if statement contains two simple conditions. The condition gender == 1 might, for example, determine whether a person is a female. The condition age >= 65 determines whether a person is a senior citizen. The two simple conditions are evaluated first because == and >= each have higher precedence than &&. The if statement then considers the combined condition gender == 1 && age >= 65, which is *true* if and only if both of the simple conditions are *true*. Finally, if this combined condition is *true*, then the preceding if statement increments seniorFemales by 1. If either or both simple conditions are *false*, the program skips the if's body and proceeds to the next statement in sequence.

The following table summarizes the && operator:

expression I	expression2	expression1 && expression2
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

The table shows all four possible combinations of zero (*false*) and nonzero (*true*) values for *expression1* and *expression2*. Such tables are often called **truth tables**. C evaluates all expressions that include relational operators, equality operators, and/or logical operators to 0 or 1. Although C sets a *true* value to 1, it accepts any nonzero value as *true*.

Logical OR (||) Operator

Now let's consider the || (logical OR) operator. Suppose we wish to ensure at some point in a program that either or both of two conditions are *true* before we choose a certain path of execution. In this case, we use the || operator, as in the following program segment:

```
if (semesterAverage >= 90 || finalExam >= 90) {
   puts("Student grade is A");
}:
```

This statement contains two simple conditions. The condition semesterAverage >= 90 determines whether the student deserves an "A" because of a solid performance throughout the semester. The condition finalExam >= 90 determines whether the student deserves an "A" because of an outstanding performance on the final exam. The if statement then considers the combined condition and awards the student an "A" if either or both of the simple conditions are *true*. The message "Student grade is A" prints unless both simple conditions are *false* (zero). The following is a truth table for the logical OR operator (||):

expression I	expression2	expression1 expression2
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

Short-Circuit Evaluation

The && operator has higher precedence than ||. Both operators associate from left-toright. An expression containing && or || operators evaluates only until it's known whether the condition is *true* or *false*. Thus, the condition

```
gender == 1 \&\& age >= 65
```

stops evaluating if gender is not equal to 1—the entire expression is guaranteed to be false. The condition continues evaluating if gender is equal to 1—the entire expression could be *true* if age is greater than or equal to 65. This performance feature for REPERF evaluating logical AND and logical OR expressions is called short-circuit evaluation.

In && expressions, make the condition that's most likely to be *false* the leftmost 📑 PERF condition. In expressions using operator ||, make the condition that's most likely to be *true* the leftmost condition. This can reduce a program's execution time.

Logical Negation (!) Operator

C provides the unary! (logical negation) operator to enable you to "reverse" the meaning of a condition. The logical negation operator has a single condition as an operand. You use it when you're interested in choosing a path of execution if the operand condition is *false*, such as in the following program segment:

```
if (!(grade == sentinelValue)) {
    printf("The next grade is %f\n", grade);
```

The parentheses around the condition grade == sentinelValue are needed because the logical negation operator has higher precedence than the equality operator. The following is a truth table for the logical negation operator:

expression	expression
0	1
nonzero	0

In most cases, you can avoid using logical negation by expressing the condition differently. For example, the preceding statement may also be written as:

```
if (grade != sentinelValue) {
    printf("The next grade is %f\n", grade);
}
```

Summary of Operator Precedence and Grouping

The following table shows the precedence and grouping of the operators introduced to this point. The operators are shown from top to bottom in decreasing order of precedence.

Operators	Grouping	Туре
++ (postfix) (postfix)	right to left	postfix
+ - ! ++ (prefix) (prefix) (type)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
H	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

The _Bool Data Type

The C standard includes a **boolean type**—represented by the keyword _Bool —which can hold only the values 0 or 1. Recall that the value 0 in a condition is *false*, while any nonzero value is *true*. Assigning any nonzero value to a _Bool sets it to 1. The standard also includes the <stdbool .h> header, which defines bool as a shorthand for the type _Bool, and true and false as named representations of 1 and 0, respectively. During preprocessing, the identifiers bool, true and false are replaced with _Bool, 1 and 0, respectively.

Self Check

[Multiple Choice] When the following if statement executes, which pair of variable values would cause seniorFemales to be incremented?

```
if (gender == 1 && age >= 65) {
    ++seniorFemales;
}
```

- a) gender is 2 and age is 60.
- b) gender is 2 and age is 73.
- c) gender is 1 and age is 19.
- d) gender is 1 and age is 65.

Answer: d.

2 (*Multiple Choice*) When the following if statement executes, which pair of variable values would not cause "Student grade is A" to print?

```
if (semesterAverage >= 90 || finalExam >= 90) {
    puts("Student grade is A");
}
```

- a) semesterAverage is 75 and finalExam is 80.
- b) semesterAverage is 85 and finalExam is 91.
- c) semesterAverage is 93 and finalExam is 67.
- d) semesterAverage is 94 and finalExam is 90.

Answer: a.

4.10 Confusing Equality (==) and Assignment (=) **Operators**



There's one type of error that C programmers, no matter how experienced, tend to make so frequently that it's worth a separate section. That error is accidentally swapping the operators == (equality) and = (assignment). What makes these swaps so damaging is the fact that they do not ordinarily cause compilation errors. Rather, statements with these errors ordinarily compile correctly, allowing programs to run to completion while likely generating incorrect results through runtime logic errors.

Two aspects of C cause these problems. One is that any expression that produces a value can be used in the decision portion of any control statement. If the value is 0, it's treated as *false*, and if the value is nonzero, it's treated as *true*. The second is that an assignment has a value—whatever value is assigned to the variable on the left of the = operator.

For example, suppose we intend to write

```
if (payCode == 4) {
         printf("%s", "You get a bonus!");
but we accidentally write
     if (payCode = 4) {
         printf("%s", "You get a bonus!");
```

The first if statement properly awards a bonus to the person whose paycode is equal to 4. The second if statement—the one with the error—evaluates the assignment expression in the if condition. The value in the condition after the assignment is 4. Because any nonzero value is *true*, the condition in this if statement is always *true*. Not only is payCode inadvertently set to 4, but the person always receives a bonus regardless of what the actual payCode is! Accidentally using operator == for assign- (X)ERR ment and accidentally using operator = for equality are both logic errors.



lvalues and rvalues

You'll probably be inclined to write conditions such as x == 7 with the variable name on the left and the constant on the right. By reversing these terms so that the constant is on the left and the variable name is on the right, as in 7 == x, if you accidentally

replace the == operator with =, you'll be protected by the compiler. The compiler will ERR (X) treat this as a syntax error because only a variable name can be placed on the left-hand side of an assignment expression. This will prevent the potential devastation of a runtime logic error.

Variable names are said to be *lvalues* (for "left values") because they can be used on the left side of an assignment operator. Constants are said to be *rvalues* (for "right values") because they can be used only on the right side of an assignment operator. An *lvalue* can also be used as an *rvalue*, but not vice versa.

Confusing == and = in Standalone Statements

The other side of the coin can be equally unpleasant. Suppose you want to assign a value to a variable with a simple statement such as

but instead write

x == 1:

ERR (X) Here, too, this is not a syntax error. The compiler evaluates the conditional expression. If x is equal to 1, the condition is *true*, and the expression returns the value 1. If x is not equal to 1, the condition is *false*, and the expression returns the value 0. Regardless of what value is returned, there's no assignment operator, so the value is simply lost. The value of x remains unaltered, probably causing an execution-time ERR (x) logic error. Unfortunately, we do not have a handy trick available to help you with this problem! Many compilers, however, will issue a warning on such a statement.



Self Check

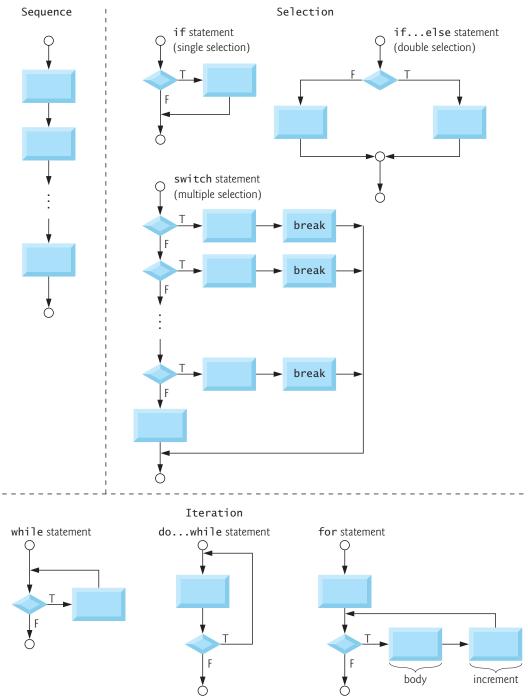
[(True/False) Accidentally swapping the operators == (equality) and = (assignment) is damaging because these errors ordinarily compile correctly, allowing programs to run to completion while likely generating incorrect results. Answer: True.

2 (*True/False*) An *rvalue* can also be used as an *lvalue*, but not vice versa. **Answer:** False. Actually, an *lvalue* can also be used as an *rvalue*, but not vice versa.

4.11 Structured-Programming Summary

Just as architects design buildings by employing the collective wisdom of their profession, programmers should design programs by employing the collective wisdom of their profession. Our field is younger than architecture, and our collective wisdom is considerably sparser. We've learned a great deal in a mere nine decades. Perhaps most important, we've learned that structured programs are easier (than unstructured programs) to understand, test, debug, modify, and even prove correct in a mathematical sense.

Chapters 3 and 4 discussed C's control statements. Now, we summarize these capabilities and introduce a simple set of rules for forming structured programs. The following diagram summarizes the control statements' flowcharts:



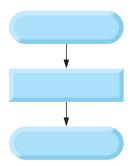
In the diagram, small circles indicate each statement's single entry point and single exit point. Connecting individual flowchart symbols arbitrarily can lead to unstructured programs. Therefore, the programming profession has chosen to combine flowchart symbols to form a limited set of control statements and to build only properly structured programs by combining control statements in two straightforward ways. For simplicity, only single-entry/single-exit control statements are used, and you may combine them only by stacking control statements in sequence or by nesting them.

Rules for Forming Structured Programs

The following table summarizes the rules for forming structured programs. We assume that the rectangle flowchart symbol indicates any action, including input/output:

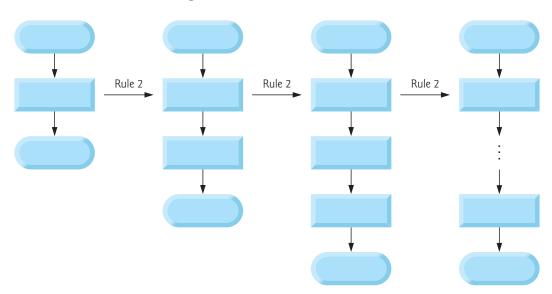
Rules for forming structured programs

- 1. Begin with the "simplest flowchart" shown in the next diagram.
- 2. "Stacking" rule—Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
- 3. "Nesting" rule—Any rectangle (action) can be replaced by any control statement (sequence, if, if...else, switch, while, do...while or for).
- 4. Rules 2 and 3 may be applied as often as you like and in *any* order.



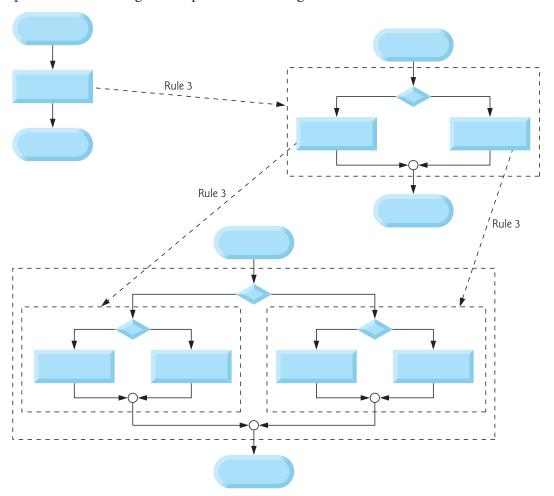
Rules for Forming Structured Programs—Stacking Rule

Applying the rules for forming structured programs always results in a structured flowchart with a neat, building-block appearance. Repeatedly applying Rule 2 to the simplest flowchart results in a structured flowchart containing many rectangles in sequence, as in the following diagram. Rule 2 generates a stack of control statements, so we call Rule 2 the stacking rule.



Rules for Forming Structured Programs—Nesting Rule

Rule 3 is called the **nesting rule**. Repeatedly applying Rule 3 to the simplest flowchart results in a flowchart with neatly nested control statements. For example, in the following diagram, the rectangle in the simplest flowchart is replaced with a double-selection (if...else) statement. Then Rule 3 is applied again to both of the rectangles in the double-selection statement, replacing each of these rectangles with double-selection statements. The dashed box around each of the double-selection statements represents the rectangle we replaced in the original flowchart.

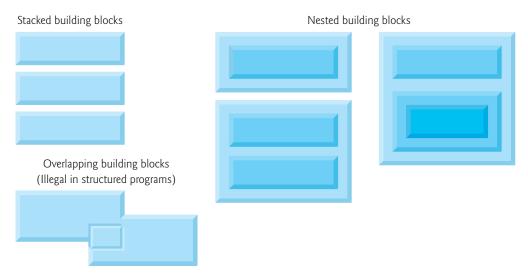


Rule 4 generates larger, more involved, and more deeply nested structures. The flowcharts that emerge from applying the rules for forming structured programs constitute the set of all possible structured flowcharts and hence the set of all possible structured programs.

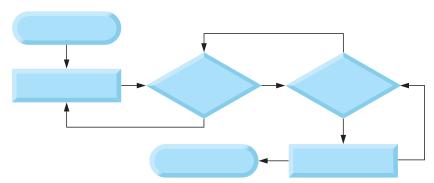
It's because of the elimination of the goto statement that these building blocks never overlap one another. The beauty of the structured approach is that we use only a small number of simple single-entry/single-exit pieces, and we assemble them in only two simple ways. The following diagram shows the kinds of stacked building blocks that emerge from applying Rule 2 and the kinds of nested building blocks that emerge from applying Rule 3. The figure also shows the kind of overlapped building

162 Chapter 4 Program Control

blocks that *cannot* appear in structured flowcharts (because of the elimination of the goto statement).



If the rules for forming structured programs are followed, an unstructured flowchart, such as the one in the following diagram, *cannot* be created:



If you're uncertain whether a particular flowchart is structured, apply the rules for forming structured programs in reverse to try to reduce the flowchart to the simplest flowchart. If you succeed, the original flowchart is structured; otherwise, it's not.

Three Forms of Control

Structured programming promotes simplicity. Böhm and Jacopini showed that only three forms of control are needed:

- Sequence.
- Selection.
- Iteration.

Sequence is straightforward. Selection is implemented in one of three ways:

- if statement (single selection).
- if...else statement (double selection).
- switch statement (multiple selection).

It's straightforward to prove that the simple if statement is sufficient to provide any form of selection. Everything that can be done with the if...else statement and the switch statement can be implemented with one or more if statements.

Iteration is implemented in one of three ways:

- while statement.
- do...while statement.
- for statement.

It's also straightforward to prove that the while statement is sufficient to provide any form of iteration. Everything that can be done with the do...while statement and the for statement can be done with the while statement.

Combining these results illustrates that any form of control ever needed in a C program can be expressed in terms of only *three* forms of control:

- sequence.
- if statement (selection).
- while statement (iteration).

And these control statements can be combined in only *two* ways—*stacking* and *nest-ing*. Indeed, structured programming promotes simplicity.

In Chapters 3 and 4, we've discussed how to compose programs from control statements containing only actions and decisions. In Chapter 5, we introduce another program-structuring unit called the function. We'll learn to compose large programs by combining functions, which, in turn, can be composed of control statements. We'll also discuss how using functions promotes software reusability.

4.12 Secure C Programming



Checking Function scanf's Return Value

Figure 4.4 used the math library function pow, which calculates the value of its first argument raised to the power of its second argument and *returns* the result as a double value. The calculation's result was then used in the statement that called pow.

Many functions return values indicating whether they executed successfully. For example, function scanf returns an int indicating whether the input operation was successful. If an input failure occurs before scanf can input a value, scanf returns the value EOF (defined in <stdio.h>); otherwise, it returns the number of items that were read into variables. If this value does *not* match the number you intended to input, then scanf was unable to complete the input operation.

Consider the following statement that expects to read one int value into grade:

```
scanf("%d", &grade); // read grade from user
```

If the user enters an integer, scanf returns 1, indicating that one value was indeed read. If the user enters a string, such as "hello", scanf returns 0, indicating that it was unable to convert the input to an integer. In this case, the variable grade does *not* receive a value.

Function scanf can input multiple values, as in

```
scanf("%d%d", &number1, &number2); // read two integers
```

If the input into both variables is successful, scanf will return 2. If the user enters a string for the first value, scanf will return 0, and neither number1 nor number2 will receive a value. If the user enters an integer followed by a string, scanf will return 1, and only number1 will receive a value.



To make your input processing more robust, check scanf's return value to ensure that the number of inputs read matches the number of inputs expected. Otherwise, your program will use the values of the variables as if scanf had completed successfully. This could lead to logic errors, program crashes or even attacks.

Range Checking

Even if a scanf operates successfully, the values read might still be invalid. For example, grades are typically integers in the range 0–100. In a program that inputs grades, you should validate each grade to ensure that it's in the range 0–100 by using range checking. You can then ask the user to reenter any value that's out of range. If a program requires inputs from a specific set of values (such as non-sequential product codes), you can ensure that each input matches a value in the set.²



Self Check

I (*Fill-In*) If an input failure occurs, scanf returns the value _____; otherwise, it returns the number of items that were read.

Answer: EOF.

2 (*Multiple Choice*) Given the following code:

```
scanf("%d%d", &grade1, &grade2); // read two integers
```

which of the following statements a), b) or c) is false?

- a) If the input is successful, scanf will return 0, indicating that integer values for both variables were input.
- b) If the user enters a string for the first value, scanf will return 0, and neither grade1 nor grade2 will receive a value.
- c) If the user enters an integer followed by a string, scanf will return 1, and only grade1 will receive a value.
- d) All of the above statements are *true*.

Answer: a) is false. Actually, if both values are input correctly, scanf will return 2.

Summary

Section 4.2 Iteration Essentials

• Most programs involve iteration (or looping). A loop is a group of instructions the computer repeatedly executes while some **loop-continuation condition** (p. 134) remains *true*.

^{2.} For more information, see Chapter 5, "Integer Security," of Robert Seacord's book *Secure Coding in C and C++, 2/e.*

- Counter-controlled iteration uses a **control variable** (p. 134) to count the number of iterations. When the correct number of iterations completes, the loop terminates, and the program resumes execution with the statement after the iteration statement.
- In sentinel-controlled iteration, a **sentinel value** is entered after all regular data items to indicate "end of data." Sentinels must be distinct from regular data items.

Section 4.3 Counter-Controlled Iteration

- Counter-controlled iteration requires the control variable's name (p. 135), its initial value (p. 135), the increment (or decrement) by which it's modified each time through the loop, and the condition that tests for the control variable's final value (p. 135).
- The control variable **increments** (or **decrements**) each time the group of instructions is performed (p. 135).

Section 4.4 for Iteration Statement

- The for iteration statement handles all the details of counter-controlled iteration.
- When a for statement begins executing, its control variable is initialized. Then, the loop-continuation condition is checked. If the condition is true, the loop's body executes. The control variable is then incremented, and the loop-continuation condition is tested. This continues until the loop-continuation condition fails.
- The general format of the for statement is

```
for (initialization; condition; increment) {
    statements
}
```

where the *initialization* expression initializes (and possibly defines) the control variable, the *condition* expression is the loop-continuation condition, and the *increment* expression increments the control variable.

- The three expressions in the for header are optional. If the *condition* is omitted, C assumes the condition is true, creating an infinite loop. One might omit the *initialization* expression if the control variable is initialized before the loop. One might omit the *increment* expression if it's calculated by statements in the for statement's body or if no increment is needed.
- The two semicolons in the for header are required.
- The "increment" may be negative to create a loop that counts downward.
- If the loop-continuation condition is initially false, the body portion of the loop isn't performed. Instead, execution proceeds with the statement following the for statement.

Section 4.5 Examples Using the for Statement

- Function pow (p. 142) performs exponentiation. The call pow(x, y) calculates the value of x raised to the yth power. The function receives two double arguments and returns a double.
- Include the header <math.h> (p. 142) whenever you need a math function such as pow.
- The conversion specification %21.2f denotes that a floating-point value will be displayed right-aligned in a field of 21 characters with two digits to the right of the decimal point.
- To left-align a value in a field, place a (minus sign) between the % and the field width.

Section 4.6 switch Multiple-Selection Statement

Occasionally, an algorithm will contain a series of decisions in which a variable or expression
is tested separately for each of the constant integral values it may assume, and different actions
are taken. This is called multiple selection. C provides the switch statement to handle this.

- Characters are normally stored in variables of type **char** (p. 146). Characters can be stored in any integer data type because they're usually represented as one-byte integers in the computer. Thus, we can treat a character as either an integer or a character, depending on its use.
- The switch statement consists of a series of case labels (p. 147), an optional default case and statements to execute for each case.
- The **getchar function** (header <stdio.h>) reads and returns as an int one character from the keyboard.
- Many computers today use the Unicode character set. ASCII is a subset of Unicode.
- Characters can be read with scanf by using the conversion specification %c.
- Assignment expressions as a whole actually have a value. This value is assigned to the variable on the left side of the =.
- EOF is often used as a sentinel value. EOF is a symbolic integer constant defined in <stdio.h>.
- On macOS/Linux systems, the E0F indicator is entered by typing Ctrl + d. On Windows, the E0F indicator can be entered by typing Ctrl + z.
- Keyword switch is followed by the **controlling expression** (p. 147) in parentheses. The value of this expression is compared with each of the case labels. If a match occurs, the statements for that case execute. If no match occurs, the default case executes.
- The **break statement** causes program control to continue with the statement after the switch. The break statement prevents the cases in a switch statement from running together.
- Each case can have one or more actions. Braces are not required around multiple actions in a case of a switch.
- Listing several case labels together executes the same set of actions for any of these cases.
- Each case in a switch statement can test only a constant integral expression (p. 149)—i.e., any combination of character constants and integer constants that evaluates to a constant integer value. A character constant can be represented as the specific character in single quotes, such as 'A'. Characters must be enclosed within single quotes to be recognized as character constants. Integer constants are simply integer values.
- In addition to integer types int and char, C provides types **short int** (which can be abbreviated as **short**) and **long int** (which can be abbreviated as **long**), as well as **unsigned** versions of all the integral types. The C standard specifies the minimum value range for each type. The actual range may be greater, depending on the implementation. For short ints, the minimum range is –32767 to +32767. The minimum range of values for long ints is –2147483647 to +2147483647. The range of values for an int is greater than or equal to that of a short int and less than or equal to that of a long int. On many of today's platforms, ints and long ints represent the same range of values. The data type signed char can be used to represent integers in the range –127 to +127 or any of the characters in the ASCII character set.

Section 4.7 do...while Iteration Statement

• The do...while statement tests the loop-continuation condition *after* the loop body is performed. Therefore, the loop body executes at least once. When a do...while terminates, execution continues with the statement after the while clause.

Section 4.8 break and continue Statements

- The **break statement**, when executed in a while, for, do...while or switch statement, immediately exits that statement. Program execution continues with the next statement.
- The **continue statement**, when executed in a while, for or do...while statement, skips the remaining statements in the body and performs the next loop iteration. The while and

do...while evaluate the loop-continuation test immediately. A for statement executes its increment expression, then tests the loop-continuation condition.

Section 4.9 Logical Operators

- Logical operators && (logical AND), || (logical OR) and ! (logical NOT, or logical negation) may be used to form complex conditions by combining simple conditions.
- The && (logical AND; p. 153) operator evaluates to *true* if and only if both of its operands are *true*.
- C evaluates all expressions that include relational operators, equality operators, and/or logical operators to 0 or 1. Although C sets a *true* value to 1, it accepts *any* nonzero value as *true*.
- The || (logical OR; p. 153) operator evaluates to true if either or both its operands are true.
- The && operator has higher precedence than ||. Both operators group from left-to-right.
- Operators && or || use short-circuit evaluation, terminating as soon as the condition is known to be false or true.
- C provides the ! (logical negation; p. 153) operator to enable you to "reverse" the meaning of a condition. Unlike the binary operators && and ||, which combine two conditions, the unary logical negation operator has only a single condition as an operand.
- The logical negation operator is placed before a condition when we're interested in choosing a path of execution if the original condition (without the logical negation operator) is *false*.

Section 4.10 Confusing Equality (==) and Assignment (=) Operators

- Programmers often accidentally swap the operators == and =. Statements with these errors ordinarily compile correctly, allowing programs to run to completion while likely generating incorrect results through runtime logic errors.
- In a condition like 7 == x, if you accidentally replace == with =, the compiler will report a syntax error. Only a variable name can be placed on the left-hand side of an assignment.
- Variable names are said to be *lvalues* (for "left values"; p. 158) because they can be used on the left side of an assignment operator.
- Constants are said to be *rvalues* (for "right values"; p. 158) because they can be used only on the right side of an assignment operator. *lvalues* can also be used as *rvalues*, but not vice versa.

Self-Review Exercises

- 4.1 Fill-In the blanks in each of the following statements.
 a) In counter-controlled iteration, a(n) ______ is used to count the number of times a group of instructions should be repeated.
 b) The _____ statement, when executed in an iteration statement, causes the next iteration of the loop to be performed immediately.
 - c) The _____ statement, when executed in an iteration statement or a switch, causes an immediate exit from the statement.
 - d) The _____ is used to test a particular variable or expression for each of the constant integral values it may assume.
- **4.2** State whether the following are *true* or *false*. If the answer is *false*, explain why.
 - a) The default case is required in the switch selection statement.
 - b) The break statement is required in a switch statement's default case.
 - c) The expression (x > y & a < b) is true if either x > y is true or a < b is true.

- d) An expression containing the || operator is *true* if either or both of its operands are *true*.
- **4.3** Write a statement or a set of statements to accomplish each of the following tasks:
 - a) Sum the odd integers between 1 and 99 using a for statement. Use the integer variables sum and count.
 - b) Print the value 333.546372 in a field width of 15 characters with precisions of 1, 2, 3, 4 and 5. Left-align the output. What are the five values that print?
 - c) Calculate the value of 2.5 raised to the power of 3 using the pow function. Print the result with a precision of 2 in a field width of 10 positions. What is the value that prints?
 - d) Print the integers from 1 to 20 using a while loop and the counter variable x. Print only five integers per line. [*Hint:* Use the calculation x % 5. When this is 0, print a newline character, otherwise print a tab character.]
 - e) Repeat Exercise 4.3(d) using a for statement.
- **4.4** Find the error in each of the following code segments and explain how to correct it:

```
a) x = 1;
  while (x \ll 10);
     ++X;
  }
b) for (double y = .1; y != 1.0; y += .1) {
     printf("%f\n", y);
c) switch (n) {
     case 1:
         puts("The number is 1");
     case 2:
         puts("The number is 2");
         break:
     default:
         puts("The number is not 1 or 2");
         break;
d) The following code should print the values 1 to 10.
    n = 1;
    while (n < 10) {
       printf("%d ", n++);
```

Answers to Self-Review Exercises

4.1 a) control variable or counter. b) continue. c) break. d) switch selection statement.

4.2 See the answers below:

- a) *False*. The default case is optional. If no default action is needed, then there's no need for a default case.
- b) *False*. The break statement is used to exit the switch statement. The break statement is not required in *any* case.
- c) *False*. Both of the relational expressions must be *true* for the entire expression to be *true* when using the && operator.
- d) True.

4.3 See the answers below:

```
a) int sum = 0;
  for (int count = 1; count <= 99; count += 2) {</pre>
     sum += count:
  }
b) printf("%-15.1f\n", 333.546372); // prints 333.5
  printf("%-15.2f\n", 333.546372); // prints 333.55
  printf("%-15.3f\n", 333.546372); // prints 333.546
  printf("%-15.4f\n", 333.546372); // prints 333.5464
  printf("%-15.5f\n", 333.546372); // prints 333.54637
c) printf("%10.2f\n", pow(2.5, 3)); // prints 15.63
d) int x = 1;
  while (x \le 20) {
     printf("%d", x);
     if (x \% 5 == 0) {
        puts("");
     }
     else {
        printf("%s", "\t");
     }
     ++X;
  }
  or
  int x = 1;
  while (x \le 20) {
     if (x \% 5 == 0) {
        printf("%d\n", x++);
     }
     else {
        printf("%d\t", x++);
     }
  }
```

```
or
  int x = 0;
  while (++x <= 20) {
     if (x \% 5 == 0) {
         printf("%d\n", x);
     }
     else {
         printf("%d\t", x);
     }
e) for (int x = 1; x \le 20; ++x) {
     printf("%d", x);
     if (x \% 5 == 0) {
         puts("");
     }
     else {
         printf("%s", "\t");
     }
  }
  or
  for (int x = 1; x \le 20; ++x) {
     if (x \% 5 == 0) {
         printf("%d\n", x);
     }
     else {
         printf("%d\t", x);
     }
```

- **4.4** a) Error: The semicolon after the while header causes an infinite loop. Correction: Replace the semicolon with a { or remove both the ; and the }.
 - b) Error: Using a floating-point number to control a for iteration statement. Correction: Use an integer, and perform the proper calculation to get the values you desire.

```
for (int y = 1; y != 10; ++y) {
    printf("%f\n", (float) y / 10);
}
```

c) Error: Missing break statement in the statements for the first case. Correction: Add a break statement at the end of the statements for the first case. This is not necessarily an error if you want the statement of case 2: to execute every time the case 1: statement executes.

d) Error: Improper relational operator used in the while iteration-continuation condition.

Correction: Use <= rather than <.

Exercises

4.5 Find the error in each of the following. (*Note:* There may be more than one error.)

```
a) For (x = 100, x >= 1, ++x) {
    printf("%d\n", x);
}
```

b) The following code should print whether a given integer is odd or even:

```
switch (value % 2) {
   case 0:
    puts("Even integer");
   case 1:
    puts("Odd integer");
}
```

c) The following code should input an integer and a character and print them. Assume the user types as input 100 A.

```
scanf("%d", &intVal);
  charVal = getchar();
  printf("Integer: %d\nCharacter: %c\n", intVal, charVal);
d) for (x = .000001; x == .0001; x += .000001) {
    printf("%.7f\n", x);
```

e) The following code should output the odd integers from 999 to 1:

```
for (x = 999; x >= 1; x += 2) {
   printf("%d\n", x);
}
```

f) The following code should output the even integers from 2 to 100:

```
counter = 2;
Do {
   if (counter % 2 == 0) {
      printf("%d\n", counter);
   }
   counter += 2;
} While (counter < 100);</pre>
```

g) The following code should sum the integers from 100 to 150 (assume total is initialized to 0):

```
for (x = 100; x <= 150; ++x); {
   total += x;
}</pre>
```

4.6 State which values of the control variable x are printed by each of the following for statements:

```
a) for (int x = 2; x <= 13; x += 2) {
    printf("%d\n", x);
}
b) for (int x = 5; x <= 22; x += 7) {
    printf("%d\n", x);
}
c) for (int x = 3; x <= 15; x += 3) {
    printf("%d\n", x);
}
d) for (int x = 1; x <= 5; x += 7) {
    printf("%d\n", x);
}
e) for (int x = 12; x >= 2; x -= 3) {
    printf("%d\n", x);
}
```

- **4.7** Write for statements that print the following sequences of values:
 - a) 1, 2, 3, 4, 5, 6, 7
 - b) 3, 8, 13, 18, 23
 - c) 20, 14, 8, 2, -4, -10
 - d) 19, 27, 35, 43, 51
- **4.8** What does the following program do?

```
#include <stdio.h>
 1
 2
    int main(void) {
 3
 4
       int x = 0;
 5
       int y = 0;
 6
 7
       // prompt user for input
       printf("%s", "Enter two integers in the range 1-20: ");
 8
       scanf("%d%d", &x, &y); // read values for x and y
 9
10
       for (int i = 1; i \leftarrow y; ++i) { // count from 1 to y
П
12
          for (int j = 1; j \le x; ++j) { // count from 1 to x
13
             printf("%s", "@");
14
15
16
          puts(""); // begin new line
17
       }
18
19
    }
```

4.9 (Sum a Sequence of Integers) Write a program that sums a sequence of integers. Assume that the first integer read with scanf specifies the number of values remaining

to be entered. Your program should read only one value each time scanf executes. A typical input sequence might be

5 100 200 300 400 500

where the 5 indicates that the next five values are to be summed.

4.10 (Average a Sequence of Integers) Write a program that calculates and prints the average of several integers. Assume the last value read with scanf is the sentinel 9999. A typical input sequence might be

10 8 11 7 9 9999

indicating that the average of all the values preceding 9999 is to be calculated.

- **4.11** *(Find the Smallest)* Write a program that finds the smallest of several integers. Assume that the first value read specifies the number of values remaining.
- **4.12** (Calculating the Sum of Even Integers) Write a program that calculates and prints the sum of the even integers from 2 to 30.
- **4.13** (*Calculating the Product of Odd Integers*) Write a program that calculates and prints the product of the odd integers from 1 to 15.
- **4.14** (*Factorials*) The *factorial* function is used frequently in probability problems. The factorial of a positive integer n (written n! and pronounced "n factorial") is equal to the product of the positive integers from 1 to n. Write a program that evaluates the factorials of the integers from 1 to 5. Print the results in tabular format. What difficulty might prevent you from calculating the factorial of 20?
- **4.15** (Modified Compound-Interest Program) Modify the compound-interest program of Section 4.5 to repeat its steps for interest rates of 5%, 6%, 7%, 8%, 9%, and 10%. Use a for loop to vary the interest rate.
- **4.16** (*Triangle-Printing Program*) Write a program that prints the following patterns separately, one below the other. Use for loops to generate the patterns. All asterisks (*) should be printed by a single printf statement of the form printf("%s", "*"); (this causes the asterisks to print side-by-side). [*Hint:* The last two patterns require that each line begin with an appropriate number of blanks.]

(A)	(B)	(C)	(D)
*	*****	*****	*
**	******	*****	**
***	*****	******	***
****	*****	*****	****
****	*****	*****	****
****	****	****	*****
****	****	****	*****
****	***	***	*****
****	**	**	******
****	*	*	******

4.17 (Calculating Credit Limits) Collecting money becomes increasingly difficult during periods of recession, so companies may tighten their credit limits to prevent

their accounts receivable (money owed to them) from becoming too large. In response to a prolonged recession, one company has cut its customers' credit limits in half. Thus, if a particular customer had a credit limit of \$2000, it's now \$1000. If a customer had a credit limit of \$5000, it's now \$2500. Write a program that analyzes the credit status of three customers of this company. For each customer you're given:

- a) The customer's account number.
- b) The customer's credit limit before the recession.
- c) The customer's current balance (i.e., the amount the customer owes).

Your program should calculate and print the new credit limit for each customer and determine (and print) which customers have balances that exceed their new credit limits.

- **4.18** (*Bar-Chart Printing Program*) One interesting application of computers is drawing graphs and bar charts. Write a program that reads five numbers (each between 1 and 30). For each number read, your program should print a line containing that number of adjacent asterisks. For example, if your program reads the number seven, it should print *******.
- **4.19** (*Calculating Sales*) An online retailer sells five different products whose retail prices are shown in the following table:

Product number	Retail price
1	\$ 2.98
2	\$ 4.50
3	\$ 9.98
4	\$ 4.49
5	\$ 6.87

Write a program that reads a series of pairs of numbers as follows:

- a) Product number.
- b) Quantity sold for one day.

Your program should use a switch statement to help determine the retail price for each product. Your program should calculate and display the total retail value of all products sold last week.

4.20 (*Truth Tables*) Complete the following truth tables by filling in each blank with 0 or 1.

Condition I	Condition2	Condition1 && Condition2
0	0	0
0	nonzero	0
nonzero	0	
nonzero	nonzero	

Condition I	Condition2	Condition1 Condition2
0	0	0
0	nonzero	1
nonzero	0	
nonzero	nonzero	

Condition I	! Condition I
0	1
nonzero	

- **4.21** Rewrite the program of Fig. 4.2 to define and initialize the variable counter before the for statement, then output the value of counter after the loop terminates.
- **4.22** (Average Grade) Modify the program of Fig. 4.5 so that it calculates the average grade for the class.
- **4.23** (Calculating the Compound Interest with Integers) Modify the program of Fig. 4.4 so that it uses only integers to calculate the compound interest. [Hint: Treat all monetary amounts as integral numbers of pennies. Then "break" the result into its dollar portion and cents portion by using the division and remainder operations, respectively. Insert a period.]
- **4.24** Assume i = 1, j = 2, k = 3 and m = 2. What does each statement print?

```
a) printf("%d", i == 1);
```

- b) printf("%d", j == 3);
- c) printf("%d", i >= 1 && j < 4);
- d) printf("%d", $m \le 99 \&\& k < m$);
- e) printf("%d", j >= i || k == m);
- f) printf("%d", $k + m < j \mid | 3 j >= k$);
- g) printf("%d", !m);
- h) printf("%d", !(j m));
- i) printf("%d", !(k > m));
- j) printf("%d", !(j > k));
- **4.25** (*Table of Decimal, Binary, Octal and Hexadecimal Equivalents*) Write a program that prints a table of the binary, octal and hexadecimal equivalents of the decimal numbers 1—256. If you're not familiar with these number systems, read online Appendix E before you attempt this exercise. [*Note:* You can display an integer as an octal or hexadecimal value with the conversion specifications %0 and %X, respectively.]
- **4.26** (Calculating the Value of π) Calculate the value of π from the infinite series

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Print a table that shows the value of π approximated by one term of this series, by two terms, by three terms, and so on. How many terms of this series do you have to use before you first get 3.14? 3.141? 3.1415? 3.14159?

4.27 (*Pythagorean Triples*) A right triangle can have sides that are all integers. The set of three integer values for a right triangle's sides is a Pythagorean triple. These three sides must satisfy the relationship that the sum of the sides' squares is equal to the hypotenuse's square. Find all Pythagorean triples for side1, side2, and the hypotenuse, all no larger than 500. Use a triple-nested for loop that tries all possibilities. This is an example of "brute-force" computing. It's not aesthetically pleasing to many people. But there are many reasons why this technique is important. First, with computing power increasing at such a phenomenal pace, solutions that would have taken years or even centuries of computer time to produce with the technology of just a few years ago can now be produced in hours, minutes, seconds or even less. Second, there are large numbers of interesting problems for which there's no known algorithmic approach other than sheer brute force. We investigate many problem-solving methodologies in this book. We'll consider brute-force approaches to various interesting problems.

4.28 (Calculating Weekly Pay) A company pays its employees as managers (who receive a fixed weekly salary), hourly workers (who receive a fixed hourly wage for up to the first 40 hours they work and "time-and-a-half" for overtime hours worked), commission workers (who receive \$250 plus 5.7% of their gross weekly sales), or pieceworkers (who receive a fixed amount of money for each of the items they produce—each pieceworker in this company works on only one type of item). Write a program to compute each employee's weekly pay. You do not know the number of employees in advance. Each type of employee has a pay code: Managers have paycode 1, hourly workers have code 2, commission workers have code 3 and pieceworkers have code 4. Use a switch to compute each employee's pay based on the paycode. Within the switch, prompt the user to enter the appropriate facts your program needs to calculate each employee's pay based on that employee's paycode. [Note: You can input values of type double using the conversion specification %1f with scanf.]

4.29 (*De Morgan's Laws*) We discussed the logical operators &&, ||, and !. De Morgan's Laws help express logical expressions more conveniently. These laws state that the expression ! (condition1 && condition2) is logically equivalent to the expression (!condition1 || !condition2). Also, the expression !(condition1 || condition2) is logically equivalent to the expression (!condition1 && !condition2). Use De Morgan's Laws to write equivalent expressions for each of the following, and then write a program to show that both the original expression and the new expression in each case are equivalent.

```
a) !(x < 5) && !(y >= 7)
b) !(a == b) || !(g != 5)
c) !((x <= 8) && (y > 4))
d) !((i > 4) || (j <= 6))
```

4.30 (*Replacing switch with if...else*) Rewrite Fig. 4.5 by replacing the switch with a nested if...else statement. Be careful to deal with the default case properly. Next,

rewrite this new version by replacing the nested if...else statement with a series of if statements. Here, too, be careful to deal with the default case properly. This exercise demonstrates that switch is a convenience and that any switch statement can be written with only single-selection statements.

4.31 (*Diamond-Printing Program*) Write a program that prints the following diamond shape. Your printf statements may print either one asterisk (*) or one blank. Use nested for statements and minimize the number of printf statements.

```
*
    **
    ***
    ****
    ****
    *****
    ****
    ****
    ***
    ***
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    *
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    *
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    *
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
    **
```

- **4.32** (*Modified Diamond-Printing Program*) Modify the program you wrote in Exercise 4.31 to read an odd number in the range 1 to 19 to specify the number of rows in the diamond. Your program should then display a diamond of the appropriate size.
- **4.33** (Roman-Numeral Equivalent of Decimal Values) Write a program that prints a table of the Roman-numeral equivalents for the decimal numbers in the range 1 to 100.
- **4.34** Describe how you'd replace a do...while loop with an equivalent while. What problem occurs when you try to replace a while loop with an equivalent do...while loop? Suppose you've been told that you must remove a while loop and replace it with a do...while. What additional control statement would you need to use? How would you use it to ensure that the resulting program behaves exactly like the original?
- **4.35** A criticism of the break and continue statements is that each is unstructured. Actually, break and continue statements can always be replaced by structured statements, though doing so can be awkward. Describe how you'd remove any break statement from a loop and replace that statement with some structured equivalent. [*Hint:* The break statement terminates a loop from the loop body. The other way to leave is by failing the loop-continuation test. Consider using in the loop-continuation test a second test that indicates "early exit because of a 'break' condition."] Use the technique you developed here to remove the break statement from the program of Fig. 4.7.
- **4.36** What does the following program segment do?

```
I for (int i = 1; i <= 5; ++i) {
2    for (int j = 1; j <= 3; ++j) {
3        for (int k = 1; k <= 4; ++k) {
4         printf("%s", "*");
5        }
6        puts("");
7    }
8    puts("");
9 }</pre>
```

- **4.37** Describe in general how you would remove any continue statement from a loop in a program and replace that statement with some structured equivalent. Use the technique you developed here to remove the continue statement from the program of Fig. 4.8.
- **4.38** ("The Twelve Days of Christmas" Song) Write a program that uses iteration and switch statements to print the song "The Twelve Days of Christmas." One switch statement should be used to print the day (i.e., "first," "second," etc.). A separate switch statement should be used to print the remainder of each verse.
- **4.39** (Limitations of Floating-Point Numbers for Monetary Amounts) Section 4.5 cautioned about using floating-point values for monetary calculations. Try this experiment: Create a float variable with the value 1000000.00. Next, add to that variable the literal float value 0.12f. Display the result using printf and the conversion specification "%.2f". What do you get?
- **4.40** (World Population Growth) World population has grown considerably over the centuries. Continued growth could eventually challenge the limits of breathable air, drinkable water, arable cropland and other limited resources. There's evidence that growth has been slowing in recent years and that world population could peak sometime this century, then start to decline.

For this exercise, research world population growth issues online. Be sure to investigate various viewpoints. Get estimates for the current world population and its growth rate (the percentage by which it's likely to increase this year). Write a program that calculates world population growth each year for the next 75 years, using the simplifying assumption that the current growth rate will stay constant. Print the results in a table. The first column should display the year from year 1 to year 75. The second column should display the anticipated world population at the end of that year. The third column should display the numerical increase in the world population that would occur that year. Using your results, determine the year in which the population would be double what it is today if this year's growth rate were to persist.

Functions



Objectives

In this chapter, you'll:

- Construct programs modularly from small pieces called functions.
- Use common math functions from the C standard library.
- Create new functions.
- Understand how function prototypes help the compiler ensure that you use functions correctly.
- Use the mechanisms that pass information between functions.
- See some commonly used C standard library headers.
- Learn how the function call and return mechanism is supported by the function-call stack and stack frames.
- Build a casino game using simulation techniques and random-number generation.
- Understand how an identifier's storage class affects its storage duration, scope and linkage.
- Write and use recursive functions, i.e., functions that call themselves.
- Continue our presentation of Secure C programming with a look at secure random-number generation.

- **5.1** Introduction
- **5.2** Modularizing Programs in C
- **5.3** Math Library Functions
- **5.4** Functions
- 5.5 Function Definitions
 - 5.5.1 **square** Function 5.5.2 maximum Function
- **5.6** Function Prototypes: A Deeper Look
- **5.7** Function-Call Stack and Stack Frames
- 5.8 Headers
- 5.9 Passing Arguments by Value and by Reference

- **5.10** Random-Number Generation
- 5.11 Random-Number Simulation Case Study: Building a Casino Game
- **5.12** Storage Classes
- **5.13** Scope Rules
- **5.14** Recursion
- **5.15** Example Using Recursion: Fibonacci Series
- **5.16** Recursion vs. Iteration
- **5.17** Secure C Programming—Secure Random-Number Generation

Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises

5.1 Introduction

Most computer programs that solve real-world problems are much larger than those presented in the first few chapters. Experience has shown that the best way to develop and maintain a program is to construct it from smaller pieces, each of which is more manageable than the original program. This technique is called divide and conquer. We'll describe some key C features for designing, implementing, operating and maintaining large programs.

5.2 Modularizing Programs in C

In C, you use functions to modularize programs by combining the new functions you write with prepackaged C standard library functions. The C standard library provides a rich collection of functions for performing common mathematical calculations, string manipulations, character manipulations, input/output and many other useful operations. Prepackaged functions make your job easier because they provide many of the capabilities you need.

The C standard includes the C language and its standard library—standard C compilers implement both. The functions printf, scanf and pow that we've used in previous chapters are from the standard library.

SE Avoid Reinventing the Wheel

Familiarize yourself with the rich collection of C standard library functions to help reduce program-development time. When possible, use standard functions instead of writing new ones. The C standard library functions are written by experts, well tested

^{1.} Some C standard library portions are designated as optional and are not available in all standard C compilers.

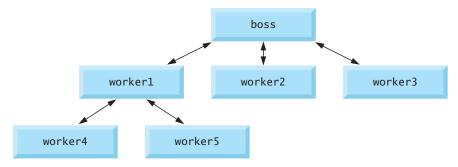
and efficient. Also, using the functions in the C standard library helps make programs per more portable.

Defining Functions

You can define functions to perform specific tasks that may be used at many points in a program. The statements defining the function are written once and are hidden from other functions. As we'll see, such hiding is crucial to good software engineering.

Calling and Returning from Functions

Functions are invoked by a function call, which specifies the function name and provides information (as arguments) that the function needs to perform its designated task. A common analogy for this is the hierarchical form of management. A boss (the calling function or caller) asks a worker (the called function) to perform a task and report back when it's done. For example, a function that displays data on the screen calls the worker function printf to perform that task. Function printf displays the data and reports back—or returns—to the caller when it completes its task. The boss function does not know how the worker function performs its designated task. The worker may call other worker functions, and the boss will be unaware of this. The following diagram shows a boss function hierarchically communicating with several worker functions:



Note that worker1 acts as a boss function to worker4 and worker5. Relationships among functions may differ from the hierarchical structure shown in this figure.

Self Check

- I (*Fill-In*) Programs are typically written by combining new functions you write with prepackaged functions available in the _____.

 Answer: C standard library.
- **2** (*Fill-In*) Functions are invoked by a function ______, which specifies the function name and provides information (as arguments) that the function needs to perform its designated task. **Answer:** call.

^{2.} In Chapter 7, Pointers, we'll discuss function pointers. You'll see that you also can call a function through a function pointer, and that you can actually pass functions to other functions.

5.3 Math Library Functions

C's math library functions (header math.h) allow you to perform common mathematical calculations. We use many of these functions in this section. To calculate and print the square root of 900.0 you might write

```
printf("%.2f", sqrt(900.0));
```

When this statement executes, it calls the math library function sqrt to calculate the square root of 900.0, then prints the result as 30.00. The sqrt function takes an argument of type double and returns a result of type double. In fact, all functions in the math library that return floating-point values return the data type double. Note that double values, like float values, can be output using the %f conversion specification. You may store a function call's result in a variable for later use as in

```
double result = sqrt(900.0);
```

Function arguments may be constants, variables, or expressions. If c = 13.0, d = 3.0 and f = 4.0, then the statement

```
printf("%.2f", sqrt(c + d * f));
```

calculates the square root of 13.0 + 3.0 * 4.0 = 25.0 and prints it as 5.00.

The following table summarizes several C math library functions. In the table, the variables x and y are of type double. The C11 standard added complex-number capabilities via the complex.h header.

Function	Description	Example
sqrt(x)	square root of x	sqrt(900.0) is 30.0 sqrt(9.0) is 3.0
cbrt(x)	cube root of x (C99 and C11 only)	cbrt(27.0) is 3.0 cbrt(-8.0) is -2.0
exp(x)	exponential function e^x	exp(1.0) is 2.718282 exp(2.0) is 7.389056
log(x)	natural logarithm of x (base e)	log(2.718282) is 1.0 log(7.389056) is 2.0
log10(x)	logarithm of x (base 10)	log10(1.0) is 0.0 log10(10.0) is 1.0 log10(100.0) is 2.0
fabs(x)	absolute value of x as a floating-point number	fabs(13.5) is 13.5 fabs(0.0) is 0.0 fabs(-13.5) is 13.5
ceil(x)	rounds x to the smallest integer not less than x	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
floor(x)	rounds x to the largest integer not greater than x	floor(9.2) is 9.0 floor(-9.8) is -10.0
pow(x, y)	x raised to power $y(x^y)$	pow(2, 7) is 128.0 pow(9, .5) is 3.0

Function	Description	Example
<pre>fmod(x, y) sin(x) cos(x) tan(x)</pre>	remainder of x/y as a floating-point number trigonometric sine of x (x in radians) trigonometric cosine of x (x in radians) trigonometric tangent of x (x in radians)	fmod(13.657, 2.333) is 1.992 sin(0.0) is 0.0 cos(0.0) is 1.0 tan(0.0) is 0.0

Self Check

- (Multiple Choice) Which of the following statements a), b) or c) is false?
 - a) You call a function by writing its name followed by its argument, or a comma-separated list of arguments, in parentheses.
 - b) The following statement calculates and stores the square root of 900.0:

```
double result = sqrt(900.0);
```

- c) To use the math library functions, you must include the math.h header.
- d) All of the above statements are *true*.

Answer: d.

2 (*True/False*) Function arguments may be constants, variables or expressions. If c = 16.0, d = 4.0 and f = 5.0, then the following statement calculates and prints the square root of 100.00:

```
printf("%.2f", sqrt(c + d * f));
```

Answer: *False.* Actually, it calculates the square root of 36.0 and prints it as 6.00.

5.4 Functions

Functions allow you to modularize a program. In programs containing many functions, main is often implemented as a group of calls to functions that perform the bulk of the program's work.

Functionalizing Programs

There are several motivations for "functionalizing" a program. The divide-and-conquer approach makes program development more manageable. Another motivation is building new programs by using existing functions. Such software reusability is a key concept in object-oriented programming languages derived from C, such as C++, Java, C# (pronounced "C sharp"), Objective-C and Swift.

With good function naming and definition, you can create programs from standardized functions that accomplish specific tasks, rather than custom code. This is known as abstraction. We use abstraction each time we use standard library functions like printf, scanf and pow. A third motivation is to avoid repeating code in a program. Packaging code as a function allows it to be executed from other program locations by calling that function.

Each function should be limited to performing a single, well-defined task, and the 🗼 SF function name should express that task. This facilitates abstraction and promotes



software reuse. If you cannot choose a concise name to describe what the function does, it may be performing too many diverse tasks. It's usually best to break such a function into smaller functions—a process called *decomposition*.

Self Check

(Fill-In) There are several motivations for "functionalizing" a program. The divide-and-conquer approach makes program development more manageable. Another motivation is ______using existing functions as building blocks to create new programs.

Answer: software reusability.

2 (True/False) Each function should perform a rich collection of related tasks, and the function name should describe those tasks.

Answer: False. Actually, each function should be limited to performing a single, welldefined task, and the function name should describe that task. This facilitates abstraction and promotes software reuse.

5.5 Function Definitions

Each program we've presented has consisted of a function called main that called standard library functions to accomplish its tasks. Now we consider how to write custom functions.

5.5.1 square Function

Consider a program that uses a function square to calculate and print the squares of the integers from 1 to 10 (Fig. 5.1).

```
| // fig05_01.c
  // Creating and using a function.
   #include <stdio.h>
4
   int square(int number); // function prototype
5
6
7
   int main(void) {
8
       // loop 10 times and calculate and output square of x each time
       for (int x = 1; x <= 10; ++x) {
9
          printf("%d ", square(x)); // function call
10
П
12
       puts("");
13
   }
14
15
    // square function definition returns the square of its parameter
   int square(int number) { // number is a copy of the function's argument
       return number * number; // returns square of number as an int
18
19
```

1 4 9 16 25 36 49 64 81 100

Fig. 5.1 | Creating and using a function. (Part 2 of 2.)

Calling Function square

Function square is invoked or called in main within the printf statement (line 10):

```
printf("%d ", square(x)); // function call
```

Function square receives a copy of the argument x's value in the parameter number (line 17). Then square calculates number * number and passes the result back to line 10 in main where square was invoked. Line 10 passes square's result to function printf, which displays the result on the screen. This process repeats 10 times—once for each iteration of the for statement.

square Function Definition

Function square's definition (lines 17–19) shows that it expects an int parameter number. The keyword int preceding the function name (line 17) indicates that square returns an integer result. The return statement in square passes the result of number * number back to the calling function.

Choosing meaningful function names and meaningful parameter names makes ASE programs more readable and helps avoid excessive comments. Programs should be written as collections of small functions. This makes programs easier to write, debug, maintain, modify and reuse.

A function requiring a large number of parameters may be performing too many 🛕 SE tasks. Consider dividing the function into smaller functions that perform the separate tasks. The function's return type, name and parameter list should fit on one line if possible.

Local Variables

All variables defined in function definitions are local variables—they can be accessed only in the function in which they're defined. Most functions have parameters that enable communicating between functions via arguments in function calls. A function's parameters are also local variables of that function.

square Function Prototype

Line 5

```
int square(int number); // function prototype
```

is a function prototype. The int in parentheses informs the compiler that square expects to receive an integer value from the caller. The int to the left of the function name square informs the compiler that square returns an integer result to the caller. Forgetting the semicolon at the end of a function prototype is a syntax error.

(X) FRR

The compiler compares square's call (line 10) to its prototype to ensure that:

• the number of arguments is correct,

- the arguments are of the correct types,
- the argument types are in the correct order, and
- the return type is consistent with the context in which the function is called.

The function prototype, first line of the function definition and function calls should all agree in the number, type and order of arguments and parameters. The function prototype and function header must have the same return type, which affects where the function can be called. For example, a function with the return type void cannot be used in an assignment statement to store a value or in a call to printf to display a value. Function prototypes are discussed in detail in Section 5.6.

Format of a Function Definition

The format of a function definition is

```
return-value-type function-name(parameter-list) {
   statements
}
```

The *function-name* is any valid identifier. The *return-value-type* is the type of the result returned to the caller. The *return-value-type* void indicates that a function does not return a value. Together, the return-value-type, function-name and parameter-list are sometimes referred to as the function header.

The *parameter-list* is a comma-separated list that specifies the parameters received by the function when it's called. If a function does not receive any parameters, *param*eter-list should contain the keyword void. Each parameter must include its type; oth-ERR (X) erwise, a compilation error occurs.

Placing a semicolon after the parameter-list's right parenthesis in a function defi-ERR (X) nition is an error, as is redefining a parameter as a local variable in a function. Although it's not incorrect to do so, do not use the same names for a function's arguments and the corresponding parameters in the function definition—this helps avoid ambiguity.

Function Body

The statements within braces form the function body, which also is a block. Local variables can be declared in any block, and blocks can be nested. Functions cannot be ERR (x) nested—defining a function inside another function is a syntax error.

Returning Control from a Function

There are three ways to return control from a called function to the point at which a function was invoked. If the function does not return a result, control is returned simply when the function-ending right brace is reached, or by executing the statement

```
return:
```

If the function does return a result, the statement

```
return expression;
```

returns the *expression*'s value to the caller.

main's Return Type

The main function's int return value indicates whether the program executed correctly. In earlier versions of C, we'd explicitly place

```
return 0; // 0 indicates successful program termination
```

at the end of main. The C standard indicates that main implicitly returns 0 if you omit the preceding statement—as we do throughout this book. You can explicitly return nonzero values from main to indicate that a problem occurred during your program's execution. For information on how to report a program failure, see the documentation for your particular operating system.

5.5.2 maximum Function

Let's consider a custom maximum function that returns the largest of three integers (Fig. 5.2). Next, they're passed to maximum (line 17), which determines the largest integer. This value is returned to main by the return statement in maximum (line 32). The printf statement in line 17 then prints the value returned by maximum.

```
// fig05_02.c
 // Finding the maximum of three integers.
 2
 3
   #include <stdio.h>
 4
 5
    int maximum(int x, int y, int z); // function prototype
 6
 7
    int main(void) {
       int number1 = 0; // first integer entered by the user
 8
 9
       int number2 = 0; // second integer entered by the user
       int number3 = 0; // third integer entered by the user
10
П
       printf("%s", "Enter three integers: ");
12
       scanf("%d%d%d", &number1, &number2, &number3);
13
14
15
       // number1, number2 and number3 are arguments
16
       // to the maximum function call
       printf("Maximum is: %d\n", maximum(number1, number2, number3));
17
    }
18
19
20
    // Function maximum definition
    int maximum(int x, int y, int z) {
21
       int max = x; // assume x is largest
22
23
24
       if (y > max) { // if y is larger than max,
25
          max = y; // assign y to max
       }
26
27
       if (z > max) { // if z is larger than max,
28
29
          max = z; // assign z to max
30
31
       return max; // max is largest value
32
33
    }
```

Fig. 5.2 | Finding the maximum of three integers. (Part 1 of 2.)

```
Enter three integers: 22 85 17
Maximum is: 85

Enter three integers: 47 32 14
Maximum is: 47

Enter three integers: 35 8 79
Maximum is: 79
```

Fig. 5.2 | Finding the maximum of three integers. (Part 2 of 2.)

The function initially assumes that its first argument (stored in the parameter x) is the largest and assigns it to max (line 22). Next, the if statement at lines 24–26 determines whether y is greater than max and, if so, assigns y to max. Then, the if statement at lines 28–30 determines whether z is greater than max and, if so, assigns z to max. Finally, line 32 returns max to the caller.

Self Check

[(Multiple Choice) The following line of code is a _____.

int square(int y);

- a) Function definition.
- b) Function statement.
- c) Function prototype.
- d) None of the above.

Answer: c.

- **2** (*Multiple Choice*) Consider the maximum function in Fig. 5.2. Which of the following statements is *false*?
 - a) The code determines the largest of three integer values.
 - b) The statement return max; sends the result back to the calling function.
 - c) The code in line 21—int maximum(int x, int y, int z)—is commonly called a function header.
 - d) If int max = x; (line 22) were accidentally replaced by int max = y; the function would still return the same result.

Answer: d) is *false*. The function would then incorrectly return the larger of the values contained in only parameters y and z.

5.6 Function Prototypes: A Deeper Look

An important C feature is the function prototype, which was borrowed from C++. The compiler uses function prototypes to validate function calls. Pre-standard C did *not* perform this kind of checking, so it was possible to call functions improperly

without the compiler detecting the errors. Such calls could result in fatal executiontime errors or nonfatal errors that caused subtle, difficult-to-detect problems. Function prototypes correct this deficiency.

You should include function prototypes for all functions to take advantage of C's ASE type-checking capabilities. Use #include preprocessor directives to obtain function prototypes from standard library headers, third-party library headers and headers for functions developed by you or your team members.

The function prototype for maximum in Fig. 5.2 (line 5) is

```
int maximum(int x, int y, int z); // function prototype
```

It states that maximum takes three arguments of type int and returns an int result. Notice that the function prototype (omitting the semicolon) is the same as the maximum definition's first line. We include parameter names in function prototypes for ASE documentation purposes. The compiler ignores these names, so the following prototype also is valid:



```
int maximum(int, int, int);
```

Compilation Errors

A function call that does not match the function prototype is a compilation error. It \(\infty \) ERR also is an error if the function prototype and the function definition disagree. For example, in Fig. 5.2, if the function prototype had been written

```
void maximum(int x, int y, int z);
```

the compiler would generate an error, because the function prototype's void return type would differ from the int return type in the function header.

Argument Coercion and "Usual Arithmetic Conversion Rules"

Another important feature of function prototypes is argument coercion, i.e., implicitly converting arguments to the appropriate type. For example, calling the math library function sqrt with an integer argument still works even though the function prototype in <math.h> specifies a double parameter. The following statement correctly evaluates sqrt(4) and prints 2.000:

```
printf("%.3f\n", sqrt(4));
```

The function prototype causes the compiler to convert a copy of the int value 4 to the double value 4.0 before passing it to sqrt. In general, argument values that do not correspond precisely to the function prototype's parameter types are converted to the proper type before the function is called. Such conversions can lead to incorrect \(\infty \mathbb{ERR} \) results if C's usual arithmetic conversion rules are not followed. These rules specify how values can be converted to other types without losing data.



In our sqrt example, an int is automatically converted to a double without changing its value—double can represent a much wider range of values than int. However, a double converted to an int truncates the double's fractional part, thus changing the original value. Converting large integer types to small integer types (e.g., long to short) can also change values.

Mixed-Type Expressions

The usual arithmetic conversion rules are handled by the compiler. They apply to mixed-type expressions—that is, expressions containing values of multiple data types. In such expressions, the compiler makes temporary copies of values that need to be converted, then converts the *copies* to the "highest" type in the expression—this is known as **promotion**. For mixed-type expressions containing at least one floating-point value:

- If one value is a long double, the other values are converted to long double.
- If one value is a double, the other values are converted to double.
- If one value is a float, the other values are converted to float.

If the mixed-type expression contains only integer types, then the usual arithmetic conversions specify a set of integer promotion rules.

Section 6.3.1 of the C standard document specifies the complete details of arithmetic operands and the usual arithmetic conversion rules. The following table lists the floating-point and integer data types with each type's printf and scanf conversion specifications. In most cases, the integer types lower in the following table are converted to higher types:

Data type	printf conversion specification	scanf conversion specification
Floating-point types		
long double	%Lf	%Lf
double	%f	%1f
float	%f	%f
Integer types		
unsigned long long int	%11u	%11u
long long int	%11d	%11d
unsigned long int	%1u	%1u
long int	%1d	%1 d
unsigned int	%u	%u
int	%d	%d
unsigned short	%hu	%hu
short	%hd	%hd
char	%с	%с

A value can be converted to a lower type only by explicitly assigning the value to a variable of lower type or by using a cast operator. Arguments are converted to the parameter types specified in a function prototype as if the arguments were being assigned to variables of those types. So, if we pass a double to our square function in Fig. 5.1, the double is converted to int (a lower type), and square usually returns an incorrect value. For example, square (4.5) returns 16, not 20.25.

ERR 🛇

Converting from a higher data type in the promotion hierarchy to a lower type can change the data value. Many compilers issue warnings in such cases.

Function Prototype Notes

If there's no function prototype for a function, the compiler forms one from the first occurrence of the function—either the function definition or a call to the function. This typically leads to warnings or errors, depending on the compiler.

(X)ERR

Always include function prototypes for the functions you define or use in your program to help prevent compilation errors and warnings.

A function prototype placed outside any function definition applies to all calls to ASE the function appearing after the function prototype. A function prototype placed in a function body applies only to calls in that function made after that prototype.



Self Check

- (Fill-In) In a mixed-type expression, the compiler makes a temporary copy of each value that needs to be converted, then converts the copies to the "highest" type in the expression—this is known as _____. **Answer:** promotion.
- 2 (Multiple Choice) Consider the following function prototype for a maximum function:

int maximum(int x, int y, int z); // function prototype

Which of the following statements is false?

- a) It states that maximum takes three arguments of type int and returns a result of type int.
- b) Parameter names are required in function prototypes.
- c) A function's prototype is often the same as the function's header, except that the header does not end in a semicolon.
- d) Forgetting the semicolon at the end of a function prototype is a syntax error. **Answer:** b) is *false*. Parameter names in function prototypes are for documentation purposes. The compiler ignores these names, so int maximum(int, int, int); is equivalent to the preceding prototype.

5.7 Function-Call Stack and Stack Frames

To understand how C performs function calls, we first need to consider a data structure (i.e., collection of related data items) known as a stack. Think of a stack as analogous to a pile of dishes. You usually place a dish at the top—referred to as pushing the dish onto the stack. Similarly, you typically remove a dish from the top—referred to as popping the dish off the stack. Stacks are known as last-in, first-out (LIFO) data structures—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

Function-Call Stack

An important mechanism for computing students to understand is the function-call stack (sometimes referred to as the program execution stack). This data structureworking "behind the scenes"—supports the function call/return mechanism. As you'll see in this section, the function-call stack also supports creating, maintaining and destroying each called function's local variables.

Stack Frames

As each function is called, it may call other functions, which may call other functions—all before any function returns. Each function eventually must return control to its caller. So, we must keep track of the return addresses that each function needs to return control to the function that called it. The function-call stack is the perfect data structure for handling this information. Each time a function calls another function, an entry is pushed onto the stack. This entry, called a **stack frame**, contains the *return address* that the called function needs in order to return to the calling function. It also contains some additional information we'll soon discuss. When a called function returns, the stack frame for the function call is popped, and control transfers to the return address specified in the popped stack frame.

Each called function always finds at the *top* of the call stack the information it needs to return to its caller. If a called function calls another function, a stack frame for the new function call is pushed onto the call stack. Thus, the return address required by the newly called function to return to its caller is now located at the top of the stack.

The stack frames have another important responsibility. Most functions have local variables, which must exist while a function is executing. They need to remain active if the function makes calls to other functions. But when a called function returns to its caller, the called function's local variables need to "go away." The called function's stack frame is a perfect place to reserve the memory for local variables. That stack frame exists only as long as the called function is active. When that function returns—and no longer needs its local variables—its stack frame is popped from the stack. Those local variables are no longer known to the program.

Stack Overflow

Of course, the amount of memory in a computer is finite, so only limited memory can be used to store stack frames on the function-call stack. If more function calls occur than can have their stack frames stored on the function-call stack, a *fatal* error known as stack overflow³ occurs.

Function-Call Stack in Action

Now let's consider how the call stack supports the operation of a square function called by main (lines 8–12 of Fig. 5.3).

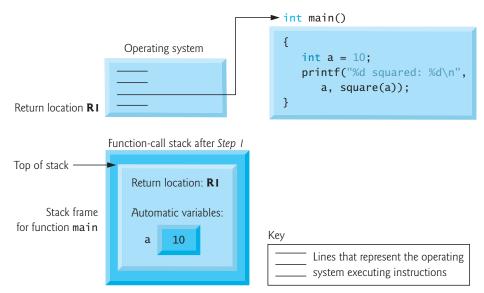
^{3.} This is how the website stackoverflow.com got its name—a popular website for getting answers to your programming questions.

```
// fig05_03.c
    // Demonstrating the function-call stack
3
   // and stack frames using a function square.
4
   #include <stdio.h>
5
6
   int square(int x); // prototype for function square
7
8
9
       int a = 10; // value to square (local variable in main)
10
       printf("%d squared: %d\n", a, square(a)); // display a squared
11
    }
12
13
    // returns the square of an integer
    int square(int x) \{ // x \text{ is a local variable } \}
       return x * x; // calculate square and return result
16
17
10 squared: 100
```

Fig. 5.3 Demonstrating the function-call stack and stack frames using a function **square**.

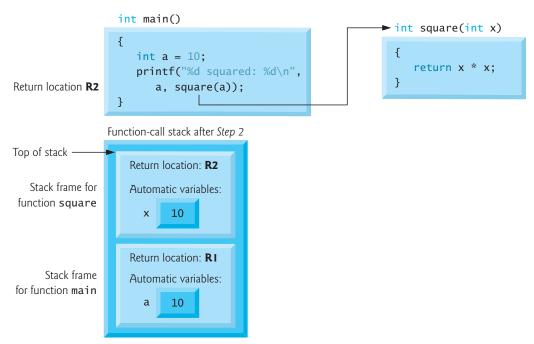
Step 1: Operating System Invokes main to Execute Application

First, the operating system calls main—this pushes a stack frame onto the stack (as shown in the following diagram). The stack frame tells main how to return to the operating system (that is, transfer to return address R1) and contains the space for main's local variable a, which is initialized to 10.



Step 2: main Invokes Function square to Perform Calculation

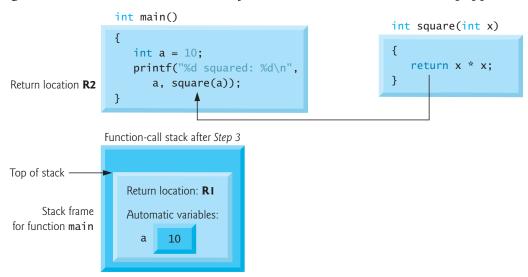
Function main—before returning to the operating system—now calls function square in line 11 of Fig. 5.3. This causes a stack frame for square (lines 15–17) to be pushed onto the function-call stack, as shown in the following diagram:



This stack frame contains the return address that square needs to return to main (i.e., R2) and the memory for square's local variable (i.e., x).

Step 3: square Returns Its Result to main

After square calculates the square of its argument, it needs to return to main—and no longer needs the memory for its local variable x. So the stack is popped—giving square the return location in main (i.e., R2) and losing square's local variable. The following diagram shows the function-call stack *after* square's stack frame has been popped:



Function main now displays the result of calling square (line 11 in Fig. 5.3). Reaching main's closing right brace pops its stack frame from the stack. This gives main the address it needs to return to the operating system (i.e., R1 in the preceding diagram). At this point, the memory for main's local variable (i.e., a) is unavailable.

Flaw in Our Discussion

There is a flaw in the preceding discussion and diagrams. We showed main calling out to square and square returning to main, but, of course, printf is a function too. As you study the code in Fig. 5.3, you might be inclined to say that main calls printf, then printf calls square. However, printf's argument values must be known in full before printf can be called. So execution proceeds as follows:

- 1. The operating system calls main, so main's stack frame is pushed onto the stack.
- 2. main calls square, so square's stack frame is pushed onto the stack.
- 3. square calculates and returns to main a value for use in printf's argument list, so square's stack frame is popped from the stack.
- **4.** main calls printf, so printf's stack frame is pushed onto the stack.
- 5. printf displays its arguments, then returns to main, so printf's stack frame is popped from the stack.
- 6. main terminates, so main's stack frame is popped from the stack.

Data Structures

You've now seen how valuable the stack data structure is in implementing a key mechanism that supports program execution. Data structures have many important applications in computer science. We discuss stacks, queues, lists and trees in Chapter 12.

/

Self Check

I (*Fill-In*) Each time a function calls another function, an entry is pushed onto the stack. This entry, called a ______, contains the return address that the called function needs in order to return to the calling function.

Answer: stack frame.

2 (*True/False*) A called function's stack frame is a perfect place to reserve the memory for local variables. That stack frame exists only as long as the called function is active. When the function returns—and no longer needs its local variables—its stack frame is popped from the stack. At that point, those local variables are no longer known to the program.

Answer: *True*.

5.8 Headers

Each standard library has a corresponding header containing the function prototypes for all the functions in that library and definitions of various data types and constants needed by those functions. The following table alphabetically lists several standard library headers that may be included in programs. The C standard includes additional headers. The term "macros"—used several times in this table—is discussed in detail in Chapter 14.

Header	Explanation			
Headers we use or discuss in this book:				
<assert.h></assert.h>	Contains information for adding diagnostics that aid program debugging.			
<ctype.h></ctype.h>	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lower-case letters to uppercase letters and vice versa.			
<float.h></float.h>	Contains the floating-point size limits of the system.			
	Contains the integral size limits of the system.			
<math.h></math.h>	Contains function prototypes for math library functions.			
<signal.h></signal.h>	Contains function prototypes and macros to handle various conditions that may arise during program execution.			
<stdarg.h></stdarg.h>	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.			
<stdio.h></stdio.h>	Contains function prototypes for the standard input/output library functions and information used by them.			
<stdlib.h></stdlib.h>	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers and other utility functions.			
<string.h></string.h>	Contains function prototypes for string-processing functions.			
<time.h></time.h>	Contains function prototypes and types for manipulating the time and date.			
Other headers:				
<errno.h></errno.h>	Defines macros that are useful for reporting error conditions.			
<locale.h></locale.h>	Contains function prototypes and other information that enables a program to be modified for the current locale on which it's running. The locale notion enables the computer system to handle different conventions for expressing data such as dates, times, currency amounts and large numbers throughout the world.			
<setjmp.h></setjmp.h>	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.			
<stddef.h></stddef.h>	Contains common type definitions used by C.			

You can create custom headers. A programmer-defined header can be included by using the #include preprocessor directive. For example, if the prototype for our square function was located in the header square.h, we'd include that header in our program by using the following directive at the top of the program:

#include "square.h"

Section 14.2 presents additional information on including headers, such as why programmer-defined headers are enclosed in quotes ("") rather than angle brackets (<>).

Self Check

I (*Fill-In*) The _____ header contains function prototypes for string-processing functions.

Answer: <string.h>.

2 (*Fill-In*) The _____ header contains information for adding diagnostics that aid program debugging.

Answer: <assert.h>.

5.9 Passing Arguments by Value and by Reference

In many programming languages, there are two ways to pass arguments—pass-by-value and pass-by-reference. When an argument is passed by value, a copy of the argument's value is made and passed to the function. Changes to the copy do not affect an original variable's value in the caller. When an argument is passed by reference, the caller allows the called function to *modify* the original variable's value.

Pass-by-value should be used whenever the called function does not need to modify the value of the caller's original variable. This prevents accidental **side** effects (variable modifications) that can hinder the development of correct and reliable software systems. Pass-by-reference should be used only with *trusted* called functions that need to modify the original variable.

In C, all arguments are passed by value. In Chapter 7, Pointers, we'll show how to achieve pass-by-reference. In Chapter 6, we'll see that array arguments are automatically passed by reference for performance reasons. We'll see in Chapter 7 that this is not a contradiction. For now, we concentrate on pass-by-value.

✓ Sel

Self Check

I (*TruelFalse*) When an argument is passed by value, a copy of the argument's value is made and passed to the function. Changes to the copy also are applied to the original variable's value in the caller.

Answer: *False.* With pass-by-value, changes to the copy do not affect an original variable's value in the caller.

2 (*True/False*) Pass-by-reference should be used only with *trusted* called functions that need to modify the original variable.

Answer: True.

5.10 Random-Number Generation

We now take a brief and, hopefully, entertaining diversion into *simulation* and *game playing*. In this and the next section, we'll develop a nicely structured game-playing program that includes multiple custom functions. The program uses functions and several of the control statements we've studied. The *element of chance* can be introduced into computer applications by using the C standard library function rand⁴ from the <stdlib.h> header.

^{4.} C standard library function rand is known to be "predictable," which can create security breach opportunities. Each of our preferred platforms offers a non-standard secure random-number generator. We'll mention these in Section 5.17, Secure C Programming—Secure Random-Number Generation.

Obtaining a Random Integer Value

Consider the following statement:

```
int value = rand();
```

The rand function generates an integer between 0 and RAND_MAX (a symbolic constant defined in the <stdlib.h> header). The C standard states that RAND_MAX's value must be at least 32,767, which is the maximum value for a two-byte (i.e., 16-bit) integer. The programs in this section were tested on Microsoft Visual C++ with a maximum RAND_MAX value of 32,767, and on GNU gcc and Xcode Clang with a maximum RAND_MAX value of 2,147,483,647. If rand truly produces integers at random, every number between 0 and RAND_MAX has an equal chance (or probability) of being chosen each time rand is called.

The range of values produced directly by rand is often different from what's needed in a specific application. For example, a program that simulates coin tossing might require only 0 for "heads" and 1 for "tails." A dice-rolling program that simulates a six-sided die would require random integers from 1 to 6.

Rolling a Six-Sided Die

To demonstrate rand, let's develop a program (Fig. 5.4) to simulate 10 rolls of a six-sided die and print each roll's value.

```
// fig05_04.c
I
   // Shifted, scaled random integers produced by 1 + rand() % 6.
2
3
   #include <stdio.h>
   #include <stdlib.h>
4
5
6
   int main(void) {
7
       for (int i = 1; i \le 10; ++i) {
8
9
          printf("%d ", 1 + (rand() % 6)); // display random die value
10
       }
11
       puts("");
12
13
   }
      5 5 6 5 1 1 5 3
```

Fig. 5.4 | Shifted, scaled random integers produced by 1 + rand() % 6.

The rand function's prototype is in <stdlib.h>. In line 9, we use the remainder operator (%) in conjunction with rand as follows

```
rand() % 6
```

to produce integers in the range 0 to 5. This is called scaling. The number 6 is called the scaling factor. We then shift the range of numbers produced by adding 1 to our previous result. The output confirms that the results are in the range 1 to 6—the order in which these random values are chosen might vary by compiler.

Rolling a Six-Sided Die 60,000,000 Times

To show that these numbers occur approximately with *equal likelihood*, let's simulate 60,000,000 rolls of a die with the program of Fig. 5.5. Each integer from 1 to 6 should appear approximately 10,000,000 times.

```
// fig05_05.c
 2
    // Rolling a six-sided die 60,000,000 times.
 3 #include <stdio.h>
  #include <stdlib.h>
 4
 5
 6
   int main(void) {
 7
       int frequency1 = 0; // rolled 1 counter
       int frequency2 = 0; // rolled 2 counter
 8
 9
       int frequency3 = 0; // rolled 3 counter
       int frequency4 = 0; // rolled 4 counter
10
       int frequency5 = 0; // rolled 5 counter
11
       int frequency6 = 0; // rolled 6 counter
12
13
14
       // loop 60000000 times and summarize results
15
       for (int roll = 1; roll <= 60000000; ++roll) {
          int face = 1 + rand() % 6; // random number from 1 to 6
16
17
          // determine face value and increment appropriate counter
18
19
          switch (face) {
20
              case 1: // rolled 1
21
                 ++frequency1;
22
                 break:
             case 2: // rolled 2
23
24
                 ++frequency2;
25
                 break:
26
             case 3: // rolled 3
                 ++frequency3;
27
28
                 break:
              case 4: // rolled 4
29
                 ++frequency4;
30
31
                 break:
             case 5: // rolled 5
32
33
                 ++frequency5;
34
                 break;
35
              case 6: // rolled 6
36
                 ++frequency6;
                 break; // optional
37
38
          }
39
       }
40
41
       // display results in tabular format
42
       printf("%s%13s\n", "Face", "Frequency");
                   1%13d\n", frequency1);
43
       printf("
       printf("
44
                   2%13d\n", frequency2);
                   3%13d\n", frequency3);
4%13d\n", frequency4);
       printf("
45
46
       printf("
```

Fig. 5.5 | Rolling a six-sided die 60,000,000 times. (Part 1 of 2.)

```
47
       printf("
                   5%13d\n", frequency5);
                   6%13d\n", frequency6);
       printf("
48
49
   }
Face
        Frequency
   1
           9999294
   2
         10002929
   3
           9995360
   4
         10000409
   5
         10005206
   6
           9996802
```

Fig. 5.5 | Rolling a six-sided die 60,000,000 times. (Part 2 of 2.)

As the program output shows, by scaling and shifting, we've used the rand function to realistically simulate the rolling of a six-sided die. Note the use of the %s conversion specification to print the character strings "Face" and "Frequency" as column headers (line 42). After we study arrays in Chapter 6, we'll show how to replace this 20-line switch statement elegantly with a single-line statement.

Randomizing the Random-Number Generator

Executing the program of Fig. 5.4 again produces

```
6 6 5 5 6 5 1 1 5 3
```

This is the exact sequence of values we showed in Fig. 5.4. How can these be random numbers? Ironically, this *repeatability* is an important characteristic of function rand. When debugging a program, this repeatability is essential for proving that corrections to a program work properly.

Function rand actually generates pseudorandom numbers. Calling rand repeatedly produces a sequence of numbers that appears to be random. However, the sequence repeats itself each time the program is executed. Once a program has been thoroughly debugged, it can be conditioned to produce a different sequence of random numbers for each execution. This is called **randomizing** and is accomplished with the standard library function **srand**. Function **srand** takes an int argument and seeds function rand to produce a different sequence of random numbers for each program execution.

We demonstrate function srand in Fig. 5.6. The function prototype for srand is found in <stdlib.h>.

```
// fig05_06.c
// Randomizing the die-rolling program.
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    printf("%s", "Enter seed: ");
```

ig. 5.6 Randomizing the die-rolling program. (Part 1 of 2.)

```
int seed = 0; // number used to seed the random-number generator
8
      scanf("%d", &seed);
9
10
       srand(seed); // seed the random-number generator
П
12
       for (int i = 1; i \le 10; ++i) {
13
         printf("%d ", 1 + (rand() % 6)); // display random die value
14
15
16
      puts("");
17
Enter seed: 67
6 1 4 6 2 1 6 1 6 4
Enter seed: 867
2 4 6 1 6 1 1 3 6 2
Enter seed: 67
```

Fig. 5.6 Randomizing the die-rolling program. (Part 2 of 2.)

6 1 4 6 2 1 6 1 6 4

Let's run the program several times and observe the results. Notice that a different sequence of random numbers is obtained each time the program is run, provided that a different seed is supplied. The first and last outputs use the same seed value, so they show the same results.

To randomize without entering a seed each time, use a statement like srand(time(NULL));

This causes the computer to read its clock to obtain the value for the seed automatically. Function time returns the number of seconds that have passed since midnight on January 1, 1970. This value is converted to an integer and used as the seed to the random-number generator. The function prototype for time is in <ti>time.h>. We'll say more about NULL in Chapter 7.

Generalized Scaling and Shifting of Random Numbers

The values produced directly by rand are always in the range:

```
0 \le rand() \le RAND_MAX
```

As you know, the following statement simulates rolling a six-sided die:

```
int face = 1 + rand() \% 6;
```

This statement always assigns an integer value (at random) to the variable face in the range $1 \le \text{face} \le 6$. The *width* of this range (i.e., the number of consecutive integers in the range) is 6, and the starting number in the range is 1. Referring to the preceding statement, we see that the width of the range is determined by the number used to

scale rand with the remainder operator (i.e., 6), and the starting number of the range is equal to the number (i.e., 1) that's added to rand % 6. We can generalize this result as follows:

```
int n = a + rand() \% b;
```

where

- a is the **shifting value** (which is equal to the first number in the desired range of consecutive integers), and
- b is the *scaling factor* (which is equal to the width of the desired range of consecutive integers).

In the exercises, you'll choose integers at random from sets of values other than ranges of consecutive integers.

\checkmark

Self Check

1 (Fill-In) ______ is an important characteristic of function rand. When we're debugging a program, this characteristic is essential for proving that corrections to a program work properly.

Answer: Repeatability.

2 (*True/False*) If rand truly produces integers at random, every number between 0 and RAND_MAX has an equal chance (or probability) of being chosen each time rand is called.

Answer: True.

3 (*Fill-In*) Once a program has been thoroughly debugged, it can be conditioned to produce a different sequence of random numbers for each execution. This is called randomizing and is accomplished with the standard library function _____. **Answer:** srand.

5.11 Random-Number Simulation Case Study: Building a Casino Game

In this section, we simulate the popular dice game known as "craps." The rules of the game are straightforward:

A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5, and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3, or 12 on the first throw (called "craps"), the player loses (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, that sum becomes the player's "point." To win, you must continue rolling the dice until you "make your point." The player loses by rolling a 7 before making the point.

Figure 5.7 simulates the game of craps and shows several sample executions.

```
1
   // fig05_07.c
    // Simulating the game of craps.
 2
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5
    #include <time.h> // contains prototype for function time
 7
    enum Status {CONTINUE, WON, LOST}; // constants represent game status
 8
 9
    int rollDice(void); // rollDice function prototype
10
    int main(void) {
11
       srand(time(NULL)); // randomize based on current time
12
13
14
       int myPoint = 0; // player must make this point to win
15
       enum Status gameStatus = CONTINUE; // may be CONTINUE, WON, or LOST
16
       int sum = rollDice(); // first roll of the dice
17
       // determine game status based on sum of dice
18
19
       switch(sum) {
20
          // win on first roll
21
          case 7: // 7 is a winner
          case 11: // 11 is a winner
22
23
             gameStatus = WON;
24
             break;
25
          // lose on first roll
          case 2: // 2 is a loser
26
27
          case 3: // 3 is a loser
          case 12: // 12 is a loser
28
             gameStatus = LOST;
29
30
             break:
          // remember point
31
32
          default:
33
             gameStatus = CONTINUE; // player should keep rolling
34
             myPoint = sum; // remember the point
35
             printf("Point is %d\n", myPoint);
36
             break; // optional
       }
37
38
39
       // while game not complete
       while (CONTINUE == gameStatus) { // player should keep rolling
40
41
          sum = rollDice(); // roll dice again
42
43
          // determine game status
          if (sum == myPoint) { // win by making point
44
45
             gameStatus = WON;
46
          }
47
          else if (7 == sum) \{ // lose by rolling 7 \}
             gameStatus = LOST;
48
49
          }
50
       }
51
```

Fig. 5.7 | Simulating the game of craps. (Part 1 of 2.)

```
52
       // display won or lost message
53
       if (WON == gameStatus) { // did player win?
54
          puts("Player wins");
55
       }
56
       else { // player lost
          puts("Player loses");
57
58
       }
    }
59
60
61
    // roll dice, calculate sum and display results
62
    int rollDice(void) {
       int die1 = 1 + (rand() % 6); // pick random die1 value
63
64
       int die2 = 1 + (rand() % 6); // pick random die2 value
65
66
       // display results of this roll
67
       printf("Player rolled %d + %d = %d\n", die1, die2, die1 + die2);
68
       return die1 + die2; // return sum of dice
69
```

Player wins on the first roll:

```
Player rolled 5 + 6 = 11
Player wins
```

Player wins on a subsequent roll:

```
Player rolled 4 + 1 = 5
Point is 5
Player rolled 6 + 2 = 8
Player rolled 2 + 1 = 3
Player rolled 3 + 2 = 5
Player wins
```

Player loses on the first roll:

```
Player rolled 1 + 1 = 2
Player loses
```

Player loses on a subsequent roll:

```
Player rolled 6 + 4 = 10
Point is 10
Player rolled 3 + 4 = 7
Player loses
```

Fig. 5.7 | Simulating the game of craps. (Part 2 of 2.)

In the game, the player must roll two dice on each roll. We define the rolldice function to roll the dice and compute and print their sum. The function is defined once, but it's called twice (lines 16 and 41). The function takes no arguments, so we've indicated void in the parameter list (line 62) and the function prototype (line 9). Function rolldice does return the sum of the two dice, so a return type of int is indicated in its function header and in its function prototype.

Enumerations

The game is reasonably involved. The player may win or lose on the first roll or any subsequent roll. Variable gameStatus, defined to be of a new type—enum Status—stores the current status. Line 7 creates a new type called an enumeration. An enumeration, introduced by the keyword enum, is a set of integer constants represented by identifiers. Enumeration constants help make programs more readable and easier to maintain. Values in an enum start with 0 and are incremented by 1. In line 7, the constant CONTINUE has the value 0, WON has the value 1 and LOST has the value 2. It's also possible to assign an integer value to each identifier in an enum (see Chapter 10). The identifiers in an enumeration must be unique, but the values may be duplicated. Use only uppercase letters in enum constant names to make them stand out in a program and to indicate they are not variables.

When the game is won, gameStatus is set to WON. When the game is lost, gameStatus is set to LOST. Otherwise, gameStatus is set to CONTINUE, and the game continues.

Game Ends on First Roll

If the game is over after the first roll, gameStatus is not CONTINUE, so the program proceeds to the if...else statement at lines 53–58, which prints "Player wins" if gameStatus is WON and "Player loses" otherwise.

Game Ends on a Subsequent Roll

After the first roll, if the game is *not* over, then sum is saved in myPoint. Execution proceeds with the while statement because gameStatus is CONTINUE. Each time through the while, rollDice is called to produce a new sum:

- If sum matches myPoint, gameStatus is set to WON, the while loop terminates, the if...else statement prints "Player wins" and execution terminates.
- If sum is 7 (line 47), gameStatus is set to LOST, the while loop terminates, the if...else statement prints "Player loses" and execution terminates.

Control Architecture

Note the program's control architecture. We've used two functions—main and roll-Dice—and the switch, while and nested if...else statements.

Related Exercises

This Building a Casino Game case study is supported by the following exercises:

- Exercise 5.47 (Craps Game Modification).
- Exercise 6.20 (Craps Game Statistics).

Self Check

I (*Fill-In*) An enumeration, introduced by the keyword ______, is a set of integer constants represented by identifiers.

Answer: enum.

(*Fill-In*) In the following statement

<pre>enum Status {CONTINUE, WON, LOST};</pre>	
the values of CONTINUE, WON and LOST are,	and
Answer: 0, 1 and 2.	

5.12 Storage Classes

In Chapters 2-4, we used identifiers for variable names. The attributes of variables include *name*, type, size and value. In this chapter, we also use identifiers as names for user-defined functions. Actually, each identifier in a program has other attributes, including storage class, storage duration, scope and linkage.

C provides the storage-class specifiers auto, register, betern and static. A storage class determines an identifier's storage duration, scope and linkage. Storage duration is the period during which an identifier exists in memory. Some exist briefly, some are repeatedly created and destroyed, and others exist for the entire program execution. Scope determines where a program can reference an identifier. Some can be referenced throughout a program, others from only portions of a program. For a multiplesource-file program, an identifier's linkage determines whether the identifier is known only in the current source file or in any source file with proper declarations. This section discusses storage classes and storage duration, and Section 5.13 discusses scope. Chapter 15 discusses identifier linkage and programming with multiple source files.

Local Variables and Automatic Storage Duration

The storage-class specifiers are split between automatic storage duration and static storage duration. The auto keyword declares that a variable has automatic storage duration. Such variables are created when program control enters the block in which they're defined. They exist while the block is active, and they're destroyed when program control exits the block.

Only variables can have automatic storage duration. A function's local variables those declared in the parameter list or function body—have automatic storage duration by default, so the auto keyword is rarely used. Automatic storage duration is a REPERF means of conserving memory because local variables exist only when they're needed. We'll refer to variables with automatic storage duration simply as local variables.

Static Storage Class

Keywords extern and static declare identifiers for variables and functions with *static* storage duration. Identifiers of static storage duration exist from the time at which the program begins execution until it terminates. For static variables, storage is allocated and initialized only once, before the program begins execution. For functions, the name of the function exists when the program begins execution. However, even though these names exist from the start of program execution, they are not always

^{5.} Keyword register is archaic and should not be used.

^{6.} C11 added the storage-class specifier _Thread_local, which is beyond this book's scope.

accessible. Storage duration and scope (where a name can be used) are separate issues, as we'll see in Section 5.13.

There are several types of identifiers with static storage duration: external identifiers (such as global variables and function names) and local variables declared with the storage-class specifier static. Global variables and function names have storage class extern by default. Global variables are created by placing variable declarations outside any function definition. They retain their values throughout program execution. Global variables and functions can be referenced by any function that follows their declarations or definitions in the file. This is one reason for using function prototypes when we include stdio.h in a program that calls printf, the function prototype is placed at the start of our file to make the name printf known to the rest of the file.

ERR(X)

Defining a variable as global rather than local allows unintended side effects to occur when a function that does not need access to the variable accidentally or maliciously modifies it. In general, you should avoid global variables except in situations PERF * with unique performance requirements (as discussed in Chapter 15). Variables used only in a particular function should be defined as local variables in that function.

> Local static variables are still known only in the function in which they're defined and retain their value when the function returns. The next time the function is called, the static local variable contains the value it had when the function last exited. The following statement declares local variable count to be static and initializes it to 1:

```
static int count = 1;
```

All numeric variables of static storage duration are initialized to zero by default if you do not explicitly initialize them.

Keywords extern and static have special meaning when explicitly applied to external identifiers. Chapter 15 discusses the explicit use of extern and static with external identifiers and multiple-source-file programs.

Self Check

I (Fill-In) Each identifier in a program has attributes, including storage class, storage duration, _____ and __ **Answer:** scope, linkage.

- **2** (*Multiple Choice*) Which of the following statements a), b) or c) is *false?*
 - a) An identifier's storage duration is the period during which the identifier exists in memory.
 - b) An identifier's scope is where the identifier can be referenced in a program.
 - c) Keyword auto declares variables of automatic storage duration. Such variables are created when program control enters the block in which they're defined. They exist while the block is active, and they're destroyed when program control exits the block.
 - d) All of the above statements are true.

Answer: d.

5.13 Scope Rules

The scope of an identifier is the portion of the program in which the identifier can be referenced. For example, a local variable in a block can be referenced only following its definition in that block or in blocks nested within that block. The four identifier scopes are function scope, file scope, block scope and function-prototype scope.

Function Scope

Labels are identifiers followed by a colon such as start:. Labels are the *only* identifiers with function scope. Labels can be used anywhere in the function in which they appear, but they cannot be referenced outside the function body. Labels are used in switch statements (as case labels) and in goto statements (see Chapter 15). Labels are hidden in the function in which they're defined. This information hiding is a means of implementing the principle of least privilege—a fundamental principle of good software engineering. In the context of an application, the principle states that code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more.

File Scope

An identifier declared outside any function has **file scope**. Such an identifier is "known" (i.e., accessible) in all functions from the point at which the identifier is declared until the end of the file. Global variables, function definitions and function prototypes placed outside a function all have file scope.

Block Scope

Identifiers defined inside a block have block scope. Block scope ends at the terminating right brace (3) of the block. Local variables defined at the beginning of a function have block scope, as do function parameters, which are considered local variables by the function. Any block may contain variable definitions. When blocks are nested and an outer block's identifier has the same name as an inner block's identifier, the outer block's identifier is hidden until the inner block terminates. While executing in the inner block, the inner block sees its local identifier's value, *not* the value of the enclosing block's identically named identifier. For this reason, you generally should avoid variable names that hide names in outer scopes. Local variables declared static still have block scope, even though they exist from before program startup. Thus, storage duration does *not* affect the scope of an identifier.

Function-Prototype Scope

The only identifiers with function-prototype scope are those used in the parameter list of a function prototype. As mentioned previously, function prototypes do not require names in the parameter list—only types are required. If a name is used in the parameter list of a function prototype, the compiler ignores it. Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity.

Scoping Example

Figure 5.8 demonstrates scoping issues with global variables, local variables and static local variables. A global variable x is defined and initialized to 1 (line 9). This global variable is hidden in any block (or function) in which a variable named x is defined. In main, a local variable x is defined and initialized to 5 (line 12). This variable is then printed to show that the global x is hidden in main. Next, a new block is defined in main with another local variable x initialized to 7 (line 17). This variable is printed to show that it hides x in the outer block of main. The variable x with value 7 is automatically destroyed when the block is exited, and the local variable x in the outer block of main is printed again to show that it's no longer hidden.

```
// fig05_08.c
 1
 2
    // Scoping.
 3
   #include <stdio.h>
 4
 5
    void useLocal(void); // function prototype
    void useStaticLocal(void); // function prototype
 7
    void useGlobal(void); // function prototype
 8
 9
    int x = 1; // global variable
10
11
    int main(void) {
12
       int x = 5; // local variable to main
13
14
       printf("local x in outer scope of main is %d\n", x);
15
16
       { // start new scope
          int x = 7; // local variable to new scope
17
18
19
          printf("local x in inner scope of main is %d\n", x);
       } // end new scope
20
21
       printf("local x in outer scope of main is %d\n", x);
22
23
24
       useLocal(); // useLocal has automatic local x
       useStaticLocal(); // useStaticLocal has static local x
25
26
       useGlobal(); // useGlobal uses global x
27
       useLocal(); // useLocal reinitializes automatic local x
       useStaticLocal(); // static local x retains its prior value
28
       useGlobal(); // global x also retains its value
29
30
31
       printf("\nlocal x in main is %d\n", x);
32
    }
33
    // useLocal reinitializes local variable x during each call
35
    void useLocal(void) {
       int x = 25; // initialized each time useLocal is called
36
37
       printf("\nlocal x in useLocal is %d after entering useLocal\n", x);
38
39
       ++X;
```

Fig. 5.8 | Scoping. (Part I of 2.)

```
printf("local x in useLocal is %d before exiting useLocal\n", x);
40
   }
41
42
43 // useStaticLocal initializes static local variable x only the first time
    // the function is called; value of x is saved between calls to this
45 // function
46  void useStaticLocal(void) {
47
       static int x = 50; // initialized once
48
       printf("\nlocal static x is %d on entering useStaticLocal\n", x);
49
50
51
       printf("local static x is %d on exiting useStaticLocal\n", x);
52
   }
53
54 // function useGlobal modifies global variable x during each call
55 void useGlobal(void) {
       printf("\nglobal x is %d on entering useGlobal\n", x);
56
57
       printf("global x is %d on exiting useGlobal\n", x);
58
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5
local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal
local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal
global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal
local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal
local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal
qlobal x is 10 on entering useGlobal
global x is 100 on exiting useGlobal
local x in main is 5
```

Fig. 5.8 | Scoping. (Part 2 of 2.)

The program defines three functions that each take no arguments and return nothing. Function useLocal defines a local variable x and initializes it to 25 (line 36). When function useLocal is called, the variable is printed, incremented, and printed again before exiting the function. Each time this function is called, the local variable x is reinitialized to 25.

Function useStaticLocal defines a static variable x and initializes it to 50 in line 47 (recall that the storage for static variables is allocated and initialized only once before the program begins execution). Local variables declared as static retain their

values even when they're out of scope. When useStaticLocal is called, x is printed, incremented, and printed again before exiting the function. In the next call to this function, the static local variable x will contain the previously incremented value 51.

Function useGlobal does not define any variables, so when it refers to variable x, the global x (line 9) is used. When useGlobal is called, the global variable is printed, multiplied by 10, and printed again before exiting the function. The next time function useGlobal is called, the global variable still has its modified value, 10. Finally, the program prints the local variable x in main again (line 31) to show that none of the function calls modified x's value because the functions all referred to variables in other scopes.



Self Check

- I (*Fill-In*) The _____ of an identifier is the portion of the program in which the identifier can be referenced. For example, a local variable in a block can be referenced only following its definition in that block or in blocks nested within that block. Answer: scope.
- **2** (*True/False*) Any block may contain variable definitions. When blocks are nested and an outer block's identifier has the same name as an inner block's identifier, the inner block's identifier is hidden until the outer block terminates.

Answer: False. Actually, when blocks are nested and an outer block's identifier has the same name as an inner block's identifier, the outer block's identifier is hidden until the inner block terminates.

5.14 Recursion

For some types of problems, it's actually useful to have functions call themselves. A **recursive function** is one that *calls itself* either directly or indirectly through another function. Recursion is a complex topic discussed at length in upper-level computer science courses. In this section and the next, we present simple recursion examples. We present an extensive treatment of recursion, which is spread throughout Chapters 5–8, 12 and 13. The table in Section 5.16 summarizes the book's recursion examples and exercises.

Base Cases and Recursive Calls

We consider recursion conceptually first, then examine several programs containing recursive functions. Recursive problem-solving approaches have several elements in common. A recursive function is called to solve a problem. The function actually knows how to solve only the simplest case(s), or so-called base case(s). If the function is called with a base case, it simply returns a result. When called with a more complex problem, the function typically divides the problem into two conceptual pieces:

- one that the function knows how to do, and
- one that it does not know how to do.

To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version. Because this new problem looks like the original problem, the function launches (calls) a fresh copy of itself to work on the smaller problem—this is referred to as a recursive call or the recursion step. The recursion step also includes a return statement, because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller.

The recursion step executes while the original call to the function is paused, waiting for the result from the recursion step. The recursion step can result in many more such recursive calls, as the function keeps dividing each problem with which it's called into two conceptual pieces. For the recursion to terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller problems must eventually *converge on the base case*. When the function recognizes the base case, it returns a result to the previous copy of the function, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result to its caller. As an example of these concepts at work, let's write a recursive program to perform a popular mathematical calculation.

Recursively Calculating Factorials

The factorial of a nonnegative integer n, written n! (pronounced "n factorial"), is the product

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

with 1! equal to 1, and 0! defined to be 1. For example, 5! is the product 5 * 4 * 3 * 2 * 1, which is equal to 120.

The factorial of an integer, number, greater than or equal to 0 can be calculated *iteratively* (nonrecursively) using a for statement as follows:

```
unsigned long long int factorial = 1;
for (int counter = number; counter > 1; --counter)
    factorial *= counter;
```

A *recursive* definition of the factorial function is arrived at by observing the following relationship:

$$n! = n \cdot (n-1)!$$

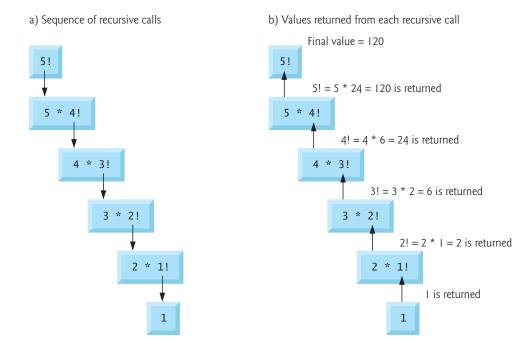
For example, 5! is clearly equal to 5 * 4! as shown by the following:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

 $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$
 $5! = 5 \cdot (4!)$

Evaluating 5! Recursively

The evaluation of 5! would proceed as shown in the following diagram. Part (a) of the following diagram shows how the succession of recursive calls proceeds until 1! is evaluated to be 1 (i.e., the *base case*), terminating the recursion. Part (b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.



Implementing Recursive Factorial Calculations

Figure 5.9 uses recursion to calculate and print the factorials of the integers 0–21 (the choice of the type unsigned long long int will be explained momentarily).

```
// fig05_09.c
 1
   // Recursive factorial function.
 2
 3 #include <stdio.h>
 5
    unsigned long long int factorial(int number);
 6
 7
   int main(void) {
 8
       // calculate factorial(i) and display result
 9
       for (int i = 0; i \le 21; ++i) {
          printf("%d! = %11u\n", i, factorial(i));
10
П
       }
    }
12
13
    // recursive definition of function factorial
14
15
    unsigned long long int factorial(int number) {
       if (number <= 1) { // base case
16
17
          return 1;
18
19
       else { // recursive step
          return (number * factorial(number - 1));
20
       }
21
22
```

Fig. 5.9 | Recursive factorial function. (Part 1 of 2.)

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 14197454024290336768
```

Fig. 5.9 | Recursive factorial function. (Part 2 of 2.)

Function factorial

The recursive factorial function first tests whether a terminating condition is true, i.e., whether number is less than or equal to 1. If number is indeed less than or equal to 1, factorial returns 1, no further recursion is necessary, and the program terminates. If number is greater than 1, the statement

```
return number * factorial(number - 1);
```

expresses the problem as the product of number and a recursive call to factorial evaluating the factorial of number - 1. The call factorial (number - 1) is a slightly simpler problem than the original calculation factorial (number).

Omitting the base case or writing the recursion step incorrectly so that it does not KRR converge on the base case will cause infinite recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution, though infinite loops do not typically exhaust memory.



Factorials Become Large Quickly

Function factorial (lines 15–22) receives an int and returns an unsigned long long int. The C standard specifies that a variable of type unsigned long long int can hold a value at least as large as 18,446,744,073,709,551,615. As can be seen in Fig. 5.9, factorial values become large quickly. We've chosen the data type unsigned long long int so the program can calculate larger factorial values. The conversion specification %11u is used to print unsigned long long int values. Unfortunately, the factorial function produces large values so quickly that even unsigned long long int does not help us print many factorial values, because that type's maximum value is quickly exceeded.

Integer Types Have Limitations

Even when we use unsigned long long int, we still can't calculate factorials beyond 21! This points to a weakness in procedural programming languages like C—the language is not easily extended to handle the unique requirements of various applications. Object-oriented languages like C++ are extensible. Through a language feature called classes, programmers can create new data types, even ones that could hold arbitrarily large integers.

Self Check

[(Fill-In) Omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case will cause ______, eventually exhausting memory.

Answer: infinite recursion.

2 (*Fill-In*) The following code should iteratively calculate the factorial of an integer, number, but the code contains a bug:

```
unsigned long long int factorial = 1;
for (int counter = number; counter >= 1; --counter)
    factorial * counter:
```

You can correct the bug by changing ______ to _____.

Answer: *, *=.

5.15 Example Using Recursion: Fibonacci Series

The Fibonacci series

```
0, 1, 1, 2, 3, 5, 8, 13, 21, \dots
```

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

The series occurs in nature and, in particular, describes a form of spiral. The ratio of successive Fibonacci numbers converges to a constant value of 1.618.... This number, too, repeatedly occurs in nature and has been called the *golden ratio* or the *golden* mean. Humans tend to find the golden mean aesthetically pleasing. Architects often design windows, rooms and buildings whose length and width are in the ratio of the golden mean. Postcards are often designed with a golden mean length/width ratio.

The Fibonacci series may be defined recursively as follows:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
```

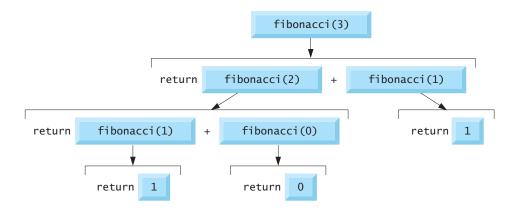
Figure 5.10 calculates the nth Fibonacci number recursively using function fibonacci. Fibonacci numbers tend to become large quickly. So, we've chosen the data type unsigned long long int for the return type in function fibonacci.

```
// fig05_10.c
    // Recursive fibonacci function.
 3
   #include <stdio.h>
 4
 5
    unsigned long long int fibonacci(int n); // function prototype
 6
 7
    int main(void) {
       // calculate and display fibonacci(number) for 0-10
 8
 9
       for (int number = 0; number <= 10; number++) {</pre>
          printf("Fibonacci(%d) = %11u\n", number, fibonacci(number));
10
11
12
13
       printf("Fibonacci(20) = %11u\n", fibonacci(20));
       printf("Fibonacci(30) = %11u\n", fibonacci(30));
14
15
       printf("Fibonacci(40) = %11u\n", fibonacci(40));
16
    }
17
18
    // Recursive definition of function fibonacci
    unsigned long long int fibonacci(int n) {
19
       if (0 == n \mid | 1 == n) \{ // base case \}
20
21
          return n;
22
23
       else { // recursive step
          return fibonacci(n - 1) + fibonacci(n - 2);
24
25
26
    }
Fibonacci(0) = 0
Fibonacci(1) = 1
Fibonacci(2) = 1
Fibonacci(3) = 2
Fibonacci(4) = 3
Fibonacci(5) = 5
Fibonacci(6) = 8
Fibonacci(7) = 13
Fibonacci(8) = 21
Fibonacci(9) = 34
Fibonacci(10) = 55
Fibonacci(20) = 6765
```

Fig. 5.10 | Recursive fibonacci function.

Fibonacci(30) = 832040 Fibonacci(40) = 102334155

The fibonacci calls from main are *not* recursive (lines 10 and 13–15), but all subsequent calls to fibonacci are recursive (line 24). Each time fibonacci is invoked, it immediately tests for the *base case*—n is equal to 0 or 1. If this is true, n is returned. Interestingly, if n is greater than 1, the recursion step generates *two* recursive calls, each a slightly simpler problem than the original call to fibonacci. The following diagram shows how function fibonacci would evaluate fibonacci(3):



Order of Evaluation of Operands

This figure raises some interesting issues about the *order* in which C compilers will evaluate operators' operands. This is a different issue from the order in which operators are applied to their operands—that is, the order dictated by the rules of operator precedence and grouping. The preceding diagram shows that while evaluating fibonacci(3), *two* recursive calls will be made, namely fibonacci(2) and fibonacci(1). But in what order will these calls be made? You might simply assume the operands will be evaluated left-to-right. For optimization reasons, C does *not* specify the order in which the operands of most operators (including +) are to be evaluated. Therefore, you should make no assumption about the order in which these calls will execute. The calls could execute fibonacci(2) first and then fibonacci(1), or the calls could execute in the reverse order, fibonacci(1) then fibonacci(2). In this and most other programs, the final result would be the same. But in some programs, the evaluation of an operand may have *side effects* that could affect the final result of the expression.

Operators for which Operand Evaluation Order Is Specified

C specifies the operand evaluation order of only four operators—&&, ||, the comma (,) operator and ?:. The first three are binary operators whose operands are guaranteed to be evaluated *left-to-right*. [Note: The commas used to separate the arguments in a function call are not comma operators.] The last operator is C's only ternary operator. Its leftmost operand is always evaluated first. If the leftmost operand evaluates to nonzero (true), the middle operand is evaluated next, and the last operand is ignored. If the leftmost operand evaluates to zero (false), the third operand is evaluated next, and the middle operand is ignored.

Exponential Complexity

A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each level of recursion in the fibonacci function has a doubling effect on the number of calls. The number of recursive calls that execute to calculate the nth Fibonacci number is "on the order of 2^n ." This rapidly gets out of hand. Calculating only the 20th Fibonacci number would require on the order of 2^{20} or about a million calls, calculating the 30th Fibonacci number would require on the

order of 2³⁰ or about a billion calls, and so on. Computer scientists refer to this as **exponential complexity**. Problems of this nature can humble even the world's most powerful computers! Complexity issues in general, and exponential complexity in particular, are discussed in detail in the upper-level computer-science course generally called "Algorithms."

The example we showed in this section used an intuitively appealing solution to calculate Fibonacci numbers, but there are better approaches. Exercise 5.48 asks you to investigate recursion in more depth and propose alternate approaches to implementing the recursive Fibonacci algorithm.

✓ Self Check

I (*True*/*False*) For optimization reasons, C specifies the order in which the operands of most operators (including +) are to be evaluated.

Answer: False. For optimization reasons, C does *not* specify the order in which the operands of most operators (including +) are to be evaluated. C specifies the order of evaluation of the operands of *only four* operators—&&, ||, the comma (,) operator and ?:.

- **2** (*Multiple Choice*) Consider the code in Fig. 5.10, which implements a recursive fibonacci function. Which of the following statements a), b) or c) is *false*?
 - a) All fibonacci calls in Fig. 5.10 are recursive calls.
 - b) Each time fibonacci is invoked, it immediately tests for the base case—n is equal to 0 or 1. If this is true, n is returned.
 - c) If n is greater than 1, the recursion step generates *two* recursive calls, each a slightly simpler problem than the original call to fibonacci.
 - d) All of the above statements are true.

Answer: a) is *false*. Actually, the calls to fibonacci from main are not recursive calls, but all subsequent calls to fibonacci are recursive (line 24).

5.16 Recursion vs. Iteration

In the previous sections, we studied two functions that can easily be implemented either recursively or iteratively. This section compares the two approaches and discusses why you might choose one approach over the other.

Common Features of Iteration and Recursion

- Both iteration and recursion are based on a *control statement*: Iteration uses an *iteration statement*; recursion uses a *selection statement*.
- Both iteration and recursion involve *repetition*: Iteration uses an *iteration statement*; recursion achieves repetition through *repeated function calls*.
- Iteration and recursion each have a *termination test*: Iteration terminates when the *loop-continuation condition fails*; recursion when a *base case is recognized*.
- Counter-controlled iteration and recursion both *gradually approach termination*: Iteration keeps modifying a counter until the counter assumes a value

- that makes the *loop-continuation condition fail*; recursion keeps producing simpler versions of the original problem until the *base case is reached*.
- Both iteration and recursion can occur *infinitely*: An *infinite loop* occurs with iteration if the loop-continuation test *never* becomes false; *infinite recursion* occurs if the recursion step does *not* reduce the problem each time in a manner that *converges on the base case*. Infinite iteration and recursion typically occur as a result of errors in a program's logic.

Negatives of Recursion

Recursion has many negatives. It *repeatedly* invokes the mechanism, and consequently the *overhead*, of function calls. This can be expensive in both processor time and memory space. Each recursive call causes *another copy* of the function (actually only the function's variables) to be created; this can *consume considerable memory*. Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted. So why choose recursion?

Recursion Is Not Required

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution may not be apparent.

Recursion Examples and Exercises Throughout This Book

Most programming textbooks introduce recursion much later than we've done here. We feel that recursion is such a sufficiently rich and complex topic that it's better to introduce it earlier and spread the examples over the remainder of the text. The following table summarizes by chapter the recursion examples and exercises in the text.

Recursion examples and exercises

Chapter 5
Factorial function
Fibonacci function
Greatest common divisor
Multiply two integers
Raising an integer to an integer power
Towers of Hanoi
Recursive main
Visualizing recursion
Chapter 6
Sum the elements of an array

Print an array

Check whether a string is a palindrome
Minimum value in an array
Linear search
Binary search
Eight Queens
Chapter 7
Maze traversal
Chapter 8
Printing a string input at the keyboard backward

Print an array backward

Print a string backward

Chapter 12
Search a linked list
Print a linked list backward
Binary tree insert
Preorder traversal of a binary
tree
Inorder traversal of a binary tree
Postorder traversal of a binary tree
Printing trees
Chapter 13
Selection sort
Quicksort

Closing Observations

Let's close this discussion with some observations that we make repeatedly throughout the book. Good software engineering is important, and high performance is important. So, we've included extensive software-engineering and performance tips throughout the book. Unfortunately, these goals are often at odds with one another. Good software engineering is key to making more manageable the task of developing the larger and more complex software systems we need. High performance is key to realizing the systems of the future that will place ever greater computing demands on hardware. Where do functions fit in here?

Software Engineering



Dividing a large program into functions promotes good software engineering. But it has a price. A heavily functionalized program—compared to a monolithic (i.e., one-piece) program without functions—makes potentially large numbers of function calls. These consume execution time on a computer's processor(s). Although monolithic programs may perform better, they're more difficult to program, test, debug, maintain and evolve.

Performance



Today's hardware architectures are tuned to make function calls efficient. C compilers help optimize your code, and today's hardware processors and multicore architecture are incredibly fast. For the vast majority of applications and software systems you'll build, concentrating on good software engineering will be more important than high-performance programming. Nevertheless, in many applications and systems, such as game programming, real-time systems, operating systems and embedded systems, performance is crucial, so we include performance tips throughout the book.

Self Check

I (*TruelFalse*) Dividing a large program into functions promotes good software engineering. But a heavily functionalized program—compared to a monolithic (i.e., one-piece) program without functions—makes a potentially large number of function calls. These consume execution time on a computer's processor(s). Although monolithic programs may perform better, they're more difficult to program, test, debug, maintain and evolve.

Answer: True.

- **2** (*Multiple Choice*) Which of the following statements is *false*?
 - a) Recursion repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both processor time and memory space.
 - b) Each recursive call causes another copy of the function's statements and variables to be created; this can consume considerable memory.

- c) Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted.
- d) A recursive approach is chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug.

Answer: b) is *false*. Actually, each recursive call causes another copy of *only the function's variables* to be created.

SEC : 5.17 Secure C Programming—Secure Random-Number Generation

In Section 5.10, we introduced the rand function for generating pseudorandom numbers. This function is sufficient for textbook examples but is not meant for use in industrial-strength applications. According to the C standard document's description of function rand, "There are no guarantees as to the quality of the random sequence produced and some implementations are known to produce sequences with distressingly non-random low-order bits." The CERT guideline MSC30-C indicates that implementation-specific random-number generation functions must be used to ensure that the random numbers produced are *not predictable*—this is extremely important, for example, in cryptography and other security applications.

The guideline presents several platform-specific random-number generators that are considered to be secure. For more information, see guideline MSC30-C at https://wiki.sei.cmu.edu/. If you're building industrial-strength applications that require random numbers, you should investigate for your platform the recommended function(s) to use. For example:

- Microsoft Windows provides the BCryptGenRandom function, which is part of Microsoft's "Cryptography API: Next Generation:"
 - https://docs.microsoft.com/en-us/windows/win32/seccng/cng-portal
- POSIX-based systems (such as Linux) provide a random function, which you
 can learn more about by executing the following command in a Terminal or
 shell:

man random

 MacOS's stdlib.h header provides the arc4random function, which you can learn more about by executing the following command in a macOS Terminal:

man arc4random

✓ Self Check

I (*True/False*) The CERT guideline MSC30-C indicates that implementation-specific random-number generation functions must be used to ensure that the random numbers produced are predictable—this is extremely important, for example, in cryptography and other security applications.

Answer: False. Actually, the CERT guideline MSC30-C indicates that implementation-specific random-number generation functions must be used to ensure that the random numbers produced are not predictable.

Summary

Section 5.1 Introduction

• The best way to develop and maintain a large program is to divide (p. 180) it into several smaller pieces, each more manageable than the original program.

Section 5.2 Modularizing Programs in C

• A function (p. 180) is invoked by a function call (p. 181), which specifies the function name and provides information (as arguments) that the function needs to perform its task.

Section 5.3 Math Library Functions

- A function is invoked by writing its name followed by a left parenthesis, the argument (or a comma-separated list of arguments) and a right parenthesis.
- Each argument may be a constant, a variable or an expression.

Section 5.4 Functions

- There are several motivations for "functionalizing" a program. The divide-and-conquer approach makes program development more manageable. Another is building new programs by using existing functions. Such software reusability is a key concept in object-oriented programming languages derived from C, such as C++, Java, C# (pronounced "C sharp"), Objective-C and Swift.
- With good function naming and definition, you can create programs from standardized functions that accomplish specific tasks, rather than custom code. This is known as abstraction. We use abstraction each time we use standard library functions like printf, scanf and pow. A third motivation is to avoid repeating code in a program. Packaging code as a function allows it to be executed from other program locations by calling that function.

Section 5.5 Function Definitions

- The arguments passed to a function should match in number, type and order with the parameters (p. 185) in the function definition.
- When a program encounters a function call, control transfers from the point of invocation to the called function, the statements of that function execute, then control returns to the caller.
- A called function can return control to the caller in one of three ways. If the function does not return a value, control is returned when the function-ending right brace is reached, or by executing the statement

```
return;
```

If the function does return a value, the statement

```
return expression;
```

returns the value of expression.

• A local variable (p. 185) is known only in a function definition. Other functions are not allowed to know the names of a function's local variables, nor is any function allowed to know the implementation details of any other function.

- A function prototype (p. 185) declares the function's name, its return type and the number, types and order of the parameters the function expects to receive.
- The general format for a function definition is

```
return-value-type function-name(parameter-list) {
    statements
}
```

If a function does not return a value, the *return-value-type* is declared as void. The *function-name* is any valid identifier. The *parameter-list* (p. 186) is a comma-separated list containing the definitions of the variables that will be passed to the function. If a function does not receive any values, *parameter-list* is declared as void.

Section 5.6 Function Prototypes: A Deeper Look

- Function prototypes enable the compiler to verify that functions are called correctly.
- The compiler ignores variable names mentioned in the function prototype.
- The C standard's usual arithmetic-conversion rules (p. 189) determine how arguments in a mixed-type expression (p. 190) are converted to the same type.

Section 5.7 Function-Call Stack and Stack Frames

- Stacks (p. 191) are known as last-in, first-out (LIFO; p. 191) data structures—the last item pushed (inserted) onto the stack is the first item popped (removed) from the stack.
- A called function must know how to return to its caller, so the return address of the calling
 function is pushed onto the program execution stack (p. 191) when the function is called. If
 a series of function calls occurs, the successive return addresses are pushed onto the stack in
 last-in, first-out order so that the last function to execute will be the first to return to its caller.
- The program execution stack contains the memory for the local variables used in each function invocation during a program's execution. This data is known as the stack frame (p. 191) of the function call. When a function call is made, the stack frame for that function call is pushed onto the program execution stack. When the function returns to its caller, the stack frame is popped off the stack and those local variables are no longer known to the program.
- If there are more function calls than can have their stack frames stored on the program execution stack, an error known as a **stack overflow** occurs.

Section 5.8 Headers

- Each standard library has a corresponding header (p. 195) containing the function prototypes for that library's functions.
- You can create and include your own headers.

Section 5.9 Passing Arguments by Value and by Reference

- When an argument is **passed by value** (p. 197), a copy is made and passed to the called function. Changes to the copy do not affect the original variable's value in the caller.
- When an argument is **passed by reference** (p. 197), the caller allows the called function to modify the original variable's value.
- All calls in C are pass-by-value by default.

Section 5.10 Random-Number Generation

• Function rand generates an integer between 0 and RAND_MAX which is defined by the C standard to be at least 32767.

- Values produced by rand can be scaled and shifted to produce values in a specific range (p. 198).
- To randomize a program, use the C standard library function srand.
- The **srand function** seeds (p. 200) the random-number generator. An srand call is ordinarily inserted in a program only after it has been thoroughly debugged. This ensures repeatability, which is essential to proving that corrections to a random-number generation program work properly.
- The function prototypes for rand and srand are contained in <stdlib.h>.
- To randomize without the need for entering a seed each time, we use srand(time(NULL)).
- The general equation for scaling and shifting a random number is

```
int n = a + rand() \% b;
```

where a is the shifting value (i.e., the first number in the desired range of consecutive integers) and b is the scaling factor (i.e., the width of the desired range of consecutive integers).

Section 5.11 Example: A Game of Chance; Introducing enum

• An enumeration (p. 206), introduced by the keyword enum, is a set of integer constants. Values in an enum start with 0 and are incremented by 1. You also can assign an integer to each identifier in an enum. The identifiers in an enumeration must be unique, but the values may be duplicated.

Section 5.12 Storage Classes

- Each identifier in a program has the attributes storage class, storage duration, scope and linkage (p. 207).
- C provides four storage classes indicated by the storage class specifiers: auto, register, extern and static (p. 207).
- An identifier's **storage duration** is when that identifier exists in memory.
- An identifier's **linkage** (p. 207) determines for a multiple-source-file program whether an identifier is known only in the current source file or in any source file with proper declarations.
- A function's local variables have **automatic storage duration** (p. 207)—they're created when program control enters the block in which they're defined, exist while the block is active and are destroyed when program control exits the block.
- Keywords **extern** and **static** declare identifiers for variables and functions of static storage duration. **Static storage duration** (p. 207) variables are allocated and initialized once, before the program begins execution.
- There are two types of identifiers with static storage duration: external identifiers (such as global variables and function names) and local variables declared with the storage-class specifier static.
- Global variables are created by placing variable definitions outside any function definition. Global variables retain their values throughout the program execution.
- Local static variables retain their value between calls to the function in which they're defined.
- All numeric variables of static storage duration are initialized to zero by default.

Section 5.13 Scope Rules

• An identifier's **scope** (p. 209) is where the identifier can be referenced in a program.

- The purpose of **information hiding** is to give functions access only to the information they need to complete their tasks. This is a means of implementing the **principle of least privilege**.
- An identifier can have function scope, file scope, block scope or function-prototype scope (p. 209).
- Labels are the only identifiers with function scope. Labels can be used anywhere in the function in which they appear but cannot be referenced outside the function body.
- An identifier declared outside any function has file scope. Such an identifier is "known" in all functions from the point at which it's declared until the end of the file.
- Identifiers defined inside a block have block scope. Block scope ends at the terminating right brace (}) of the block.
- Local variables have block scope, as do function parameters, which are local variables.
- Any block may contain variable definitions. When blocks are nested, and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is "hidden" until the inner block terminates.
- The only identifiers with function-prototype scope are those used in the parameter list of a function prototype.

Section 5.14 Recursion

- A recursive function (p. 212) is a function that calls itself either directly or indirectly.
- If a recursive function is called with a **base case** (p. 212), the function simply returns a result. If it's called with a more complex problem, it divides the problem into two conceptual pieces: a piece that the function knows how to do and a slightly smaller version of the original problem. Because this new problem looks like the original, the function launches a recursive call to work on the smaller problem.
- For recursion to terminate, each time the recursive function calls itself with a slightly simpler version of the original problem, the sequence of smaller and smaller problems must converge on the **base case**. When the function recognizes the base case, the result is returned to the previous function call, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result.
- Standard C does not specify the order in which the operands of most operators (including +) are to be evaluated. Of C's many operators, the standard specifies the order of evaluation of the operands of only the operators &&, ||, the comma (,) operator and ?:. The first three of these are binary operators whose two operands are evaluated left-to-right. The last operator is C's only ternary operator. Its leftmost operand is evaluated first; if it evaluates to nonzero, the middle operand is evaluated next and the last operand is ignored; if the leftmost operand evaluates to zero, the third operand is evaluated next and the middle operand is ignored.

Section 5.16 Recursion vs. Iteration

- Both iteration and recursion are based on a control structure: Iteration uses an iteration statement; recursion uses a selection statement.
- Both iteration and recursion involve repetition: Iteration uses an iteration statement; recursion achieves repetition through repeated function calls.
- Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.

- Iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on the base case.
- Recursion repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both processor time and memory space.

Self-Review Exercises

5.1	Answer each of the following:			
	a) are used to modularize programs.			
	b) A function is invoked with a(n)			
	c) A variable known only within the function in which it's defined is called			
	a(n)			
	d) The statement is used to pass an expression's value back to a calling function.			
	e) Keyword is used in a function header to indicate that a function			
	does not return a value or to indicate that a function contains no parameters.			
	f) The of an identifier is the portion of the program in which the			
	identifier can be used.			
	g) The three ways to return control from a called function to a caller are			
	, and			
	h) A(n) allows the compiler to check the number, types, and order of			
	the arguments passed to a function.			
	i) The function is used to produce random numbers.			
	j) The function is used to set the random number seed to randomize			
	a program.			
	k) The storage-class specifiers are,, and			
	l) Variables declared in a block or in the parameter list of a function have storage class, unless specified otherwise.			
	m)A non-static variable defined outside any block or function is a(n) variable.			
	n) For a local variable in a function to retain its value between calls to the function, it must be declared with the storage-class specifier.			
	o) The four identifier scopes are,, and			
	p) A function that calls itself either directly or indirectly is a(n) function.			
	q) A recursive function typically has two components: one that provides a means for the recursion to terminate by testing for a(n) case, and one that expresses the problem as a recursive call for a slightly simpler problem than the original call.			

5.2 Consider the following program:

```
#include <stdio.h>
int cube(int y);
```

```
4 int main(void) {
5    for (int x = 1; x <= 10; ++x) {
6        printf("%d\n", cube(x));
7    }
8    }
9
10 int cube(int y) {
11    return y * y * y;
12 }</pre>
```

State the scope (function scope, file scope, block scope or function-prototype scope) of each of the following elements:

- a) The variable x in main.
- b) The variable y in cube.
- c) The function cube.
- d) The function main.
- e) The function prototype for cube.
- f) The identifier y in the function prototype for cube.
- **5.3** Write a program that tests whether the examples of the math library function calls shown in the table of Section 5.3 actually produce the indicated results.
- **5.4** Give the function header for each of the following functions:
 - a) Function hypotenuse that takes two double arguments, side1 and side2, and returns a double result.
 - b) Function smallest that takes three integers, x, y, z, and returns an integer.
 - c) Function instructions that does not receive any arguments and does not return a value.
 - d) Function intToFloat that takes an integer argument, number, and returns a float.
- **5.5** Give the function prototype for each of the following:
 - a) The function described in Exercise 5.4(a).
 - b) The function described in Exercise 5.4(b).
 - c) The function described in Exercise 5.4(c).
 - d) The function described in Exercise 5.4(d).
- **5.6** Write a declaration for floating-point variable lastValue that's to retain its value between calls to the function in which it's defined.
- **5.7** Find the error in each of the following program segments and explain how the error can be corrected (see also Exercise 5.46):

```
a) int g(void) {
    printf("%s", "Inside function g\n");
    int h(void) {
       printf("%s", "Inside function h\n");
    }
}
```

```
b) int sum(int x, int y) {
     int result = x + y;
c) void f(float a); {
     float a:
     printf("%f", a);
d) int sum(int n) {
     if (0 == n) {
         return 0;
     }
     else {
        n + sum(n - 1);
     }
  }
e) void product(void) {
     printf("%s", "Enter three integers: ")
     int a:
     int b:
     int c;
     scanf("%d%d%d", &a, &b, &c);
     int result = a * b * c;
     printf("Result is %d", result);
     return result:
  }
```

Answers to Self-Review Exercises

- **5.1** a) functions. b) function call. c) local variable. d) return. e) void. f) scope. g) return; or return expression; or encountering the closing right brace of a function. h) function prototype. i) rand. j) srand. k) auto, register, extern, static. l) auto. m) external, global. n) static. o) function scope, file scope, block scope, function-prototype scope. p) recursive. q) base.
- **5.2** a) Block scope. b) Block scope. c) File scope. d) File scope. e) File scope. f) Function-prototype scope.
- **5.3** See below. [*Note:* On most Linux systems, you must use the -1m option when compiling this program.]

```
// ex05_03.c
// Testing the math library functions
#include <stdio.h>
#include <math.h>
```

```
6
    int main(void) {
 7
       // calculates and outputs the square root
 8
       printf("sqrt(%.1f) = %.1f\n", 900.0, sqrt(900.0));
 9
       printf("sqrt(%.1f) = %.1f\n", 9.0, sqrt(9.0));
10
11
       // calculates and outputs the cube root
12
       printf("cbrt(%.1f) = \%.1f\n", 27.0, cbrt(27.0));
       printf("cbrt(%.1f) = %.1f\n", -8.0, cbrt(-8.0));
13
14
15
       // calculates and outputs the exponential function e to the x
       printf("exp(%.1f) = %f\n", 1.0, exp(1.0));
16
17
       printf("exp(%.1f) = %f \ ". 2.0, exp(2.0));
18
       // calculates and outputs the logarithm (base e)
19
20
       printf("log(%f) = %.1f\n", 2.718282, log(2.718282));
21
       printf("log(%f) = %.1f\n", 7.389056, log(7.389056));
22
23
       // calculates and outputs the logarithm (base 10)
       printf("log10(%.1f) = %.1f\n", 1.0, log10(1.0));
24
25
       printf("log10(\%.1f) = \%.1f\n", 10.0, log10(10.0));
26
       printf("log10(\%.1f) = \%.1f\n", 100.0, log10(100.0));
27
28
       // calculates and outputs the absolute value
29
       printf("fabs(%.1f) = %.1f\n", 13.5, fabs(13.5));
       printf("fabs(%.1f) = %.1f\n", 0.0, fabs(0.0));
printf("fabs(%.1f) = %.1f\n", -13.5, fabs(-13.5));
30
31
32
33
       // calculates and outputs ceil(x)
34
       printf("ceil(%.1f) = %.1f\n", 9.2, ceil(9.2));
35
       printf("ceil(%.1f) = %.1f\n", -9.8, ceil(-9.8));
36
37
       // calculates and outputs floor(x)
38
       printf("floor(%.1f) = %.1f \ ", 9.2, floor(9.2));
       printf("floor(%.1f) = \%.1f\n", -9.8, floor(-9.8));
39
40
41
       // calculates and outputs pow(x, y)
       printf("pow(%.1f, %.1f) = %.1f\n", 2.0, 7.0, pow(2.0, 7.0));
42
43
       printf("pow(%.1f, %.1f) = %.1f\n", 9.0, 0.5, pow(9.0, 0.5));
44
45
       // calculates and outputs fmod(x, y)
46
       printf("fmod(%.3f, %.3f) = %.3f \ n", 13.657, 2.333,
47
          fmod(13.657, 2.333));
48
49
       // calculates and outputs sin(x)
50
       printf("sin(\%.1f) = \%.1f \setminus n", 0.0, sin(0.0));
51
52
       // calculates and outputs cos(x)
53
       printf("cos(\%.1f) = \%.1f\n", 0.0, cos(0.0));
54
55
       // calculates and outputs tan(x)
56
       printf("tan(%.1f) = %.1f\n", 0.0, tan(0.0));
57
   }
```

```
sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
cbrt(27.0) = 3.0
cbrt(-8.0) = -2.0
\exp(1.0) = 2.718282
\exp(2.0) = 7.389056
\log(2.718282) = 1.0
log(7.389056) = 2.0
\log 10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(13.657, 2.333) = 1.992
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0
```

- **5.4** See the answers below:
 - a) double hypotenuse(double side1, double side2)
 - b) int smallest(int x, int y, int z)
 - c) void instructions(void)
 - d) float intToFloat(int number)
- **5.5** See the answers below:
 - a) double hypotenuse(double side1, double side2);
 - b) int smallest(int x, int y, int z);
 - c) void instructions(void);
 - d) float intToFloat(int number);
- **5.6** static float lastValue;
- **5.7** See the answers below:
 - a) Error: Function h is defined in function g.
 Correction: Move the definition of h out of the definition of g.
 - b) Error: The function body is supposed to return an integer, but does not. Correction: Replace the statement in the function body with:

```
return x + y;
```

c) Error: Semicolon after the right parenthesis that encloses the parameter list, and redefining the parameter a in the function definition.

Correction: Delete the semicolon after the right parenthesis of the parameter list, and delete the declaration float a; in the function body.

d) Error: n + sum(n - 1) is not returned; sum returns an improper result.

Correction: Rewrite the statement in the else clause as

```
return n + sum(n - 1);
```

e) Error: The function returns a value when it's not supposed to. Correction: Eliminate the return statement.

Exercises

5.8 Show the value of x after each of the following statements is performed:

```
a) x = fabs(7.5);
b) x = floor(7.5);
c) x = fabs(0.0);
d) x = ceil(0.0);
e) x = fabs(-6.4);
f) x = ceil(-6.4);
g) x = ceil(-fabs(-8 + floor(-5.5)));
```

5.9 (Parking Charges) A parking garage charges a \$2.00 minimum fee to park for up to three hours and an additional \$0.50 per hour for each hour or part thereof over three hours. The maximum charge for any given 24-hour period is \$10.00. Assume that no car parks for longer than 24 hours at a time. Write a program that calculates and prints the parking charges for each of three customers who parked their cars in this garage yesterday. You should enter the hours parked for each customer. Your program should print the results in a tabular format, and should calculate and print the total of yesterday's receipts. The program should use the function calculateCharges to determine the charge for each customer. Your outputs should appear in the following format:

```
Car Hours Charge
1 1.5 2.00
2 4.0 2.50
3 24.0 10.00
TOTAL 29.5 14.50
```

5.10 (*Rounding Numbers*) An application of function floor is rounding a value to the nearest integer. The statement

```
y = floor(x + .5);
```

rounds x to the nearest integer and assigns the result to y. Write a program that reads several numbers and rounds each of these numbers to the nearest integer. For each number processed, print both the original number and the rounded number.

5.11 (*Rounding Numbers*) Function floor may be used to round a number to a specific decimal place. The statement

```
y = floor(x * 10 + .5) / 10;
```

rounds x to the tenths position (the first position to the right of the decimal point). The statement

```
y = floor(x * 100 + .5) / 100;
```

rounds x to the hundredths position (the second position to the right of the decimal point). Write a program that defines functions to round a number x in various ways:

- a) roundToInteger(number)
- b) roundToTenths(number)
- c) roundToHundreths(number)
- d) roundToThousandths(number)

For each value the program inputs, display the original value, the number rounded to the nearest integer, the number rounded to the nearest tenth, the number rounded to the nearest hundredth, and the number rounded to the nearest thousandth.

- **5.12** Answer each of the following questions:
 - a) What does it mean to choose numbers "at random"?
 - b) Why is the rand function useful for simulating games of chance?
 - c) Why would you randomize a program by using srand? Under what circumstances is it desirable not to randomize?
 - d) Why is it often necessary to scale and/or shift the values produced by rand?
- **5.13** Write statements that assign random integers to the variable n in the following ranges:
 - a) $1 \le n \le 2$
 - b) $1 \le n \le 100$
 - c) $0 \le n \le 9$
 - d) $1000 \le n \le 1112$
 - $e) -1 \le n \le 1$
 - f) $-3 \le n \le 11$
- **5.14** For each of the following sets of integers, write a single statement that will print a number at random from the set:
 - a) 2, 4, 6, 8, 10.
 - b) 3, 5, 7, 9, 11.
 - c) 6, 10, 14, 18, 22.
- **5.15** (*Hypotenuse Calculations*) Define a function called hypotenuse that calculates a right triangle's hypotenuse, based on the values of the other two sides. The function should take two double arguments and return the hypotenuse as a double. Test your program with the side values specified in the following table:

Side I	Side 2
3.0	4.0
5.0	12.0
8.0	15.0

5.16 (*Exponentiation*) Write a function integerPower(base, exponent) that returns the value of

```
baseexponent
```

For example, integerPower(3, 4) = 3 * 3 * 3 * 3. Assume that exponent is a positive, nonzero integer, and base is an integer. Function integerPower should use a for statement to control the calculation. Do not use any math library functions.

- **5.17** (*Multiples*) Write a function isMultiple that determines for a pair of integers whether the second integer is a multiple of the first. The function should take two integer arguments and return 1 (true) if the second is a multiple of the first, and 0 (false) otherwise. Use this function in a program that inputs a series of pairs of integers.
- **5.18** (Even or Odd) Write a program that inputs a series of integers and passes them one at a time to function is Even, which uses the remainder operator to determine whether an integer is even. The function should take an integer argument and return 1 if the integer is even and 0 otherwise.
- **5.19** (*Square of Asterisks*) Write a function that displays a solid square of asterisks whose side is specified in integer parameter side. For example, if side is 4, the function displays:

```
****

****

****
```

5.20 (*Displaying a Square of Any Character*) Modify the function in Exercise 5.19 to form the square out of whatever character is contained in char parameter fillCharacter. Thus if side is 5 and fillCharacter is "#", then this function should print:

```
#####
#####
#####
#####
```

- **5.21** (*Project: Drawing Shapes with Characters*) Use techniques similar to those developed in Exercises 5.19 and 5.20 to produce a program that graphs a wide range of shapes.
- **5.22** (Separating Digits) Write program segments to accomplish each of the following:
 - a) Calculate the int part of the quotient when int a is divided by int b.
 - b) Calculate the int remainder when int a is divided by int b.
 - c) Use the program pieces developed in a) and b) to write a function that inputs an integer between 1 and 32767 and prints it as a series of digits, with two spaces between each digit. For example, 4562 should be printed as:

- **5.23** (*Time in Seconds*) Write a function that takes the time as three integer arguments (for hours, minutes and seconds) and returns the number of seconds since the last time the clock "struck 12." Use this function to calculate the amount of time in seconds between two times, both of which are within one 12-hour cycle of the clock.
- **5.24** (*Temperature Conversions*) Implement the following integer functions:
 - a) toCelsius returns the Celsius equivalent of a Fahrenheit temperature.
 - b) toFahrenheit returns the Fahrenheit equivalent of a Celsius temperature.

Use these functions to write a program that prints charts showing the Fahrenheit equivalents of all Celsius temperatures from 0 to 100 degrees, and the Celsius equivalents of all Fahrenheit temperatures from 32 to 212 degrees. Print the outputs in a tabular format that minimizes the number of lines of output while remaining readable.

- **5.25** *(Find the Minimum)* Write a function that returns the smallest of three floating-point numbers.
- **5.26** (*Perfect Numbers*) An integer number is said to be a *perfect number* if its factors, including 1 (but not the number itself), sum to the number. For example, 6 is a perfect number because 6 = 1 + 2 + 3. Write a function is Perfect that determines whether parameter number is a perfect number. Use this function in a program that determines and prints all the perfect numbers between 1 and 1000. Print the factors of each perfect number to confirm that the number is indeed perfect. Challenge the power of your computer by testing numbers much larger than 1000.
- **5.27** (*Prime Numbers*) An integer is said to be *prime* if it's divisible by only 1 and itself. For example, 2, 3, 5 and 7 are prime, but 4, 6, 8 and 9 are not. Write a function that determines whether a number is prime. Use this function in a program that determines and prints all the prime numbers between 1 and 10,000. How many of these 10,000 numbers do you really have to test before being sure that you have found all the primes? Initially you might think that n/2 is the upper limit for which you must test to see whether a number is prime, but you need go only as high as the square root of n. Rewrite the program, and run it both ways. Estimate the performance improvement.
- **5.28** (*Reversing Digits*) Write a function that takes an integer value and returns the number with its digits reversed. For example, given the number 7631, the function should return 1367.
- **5.29** (Greatest Common Divisor) The greatest common divisor (GCD) of two integers is the largest integer that evenly divides each of the two numbers. Write a function gcd that returns the greatest common divisor of two integers.
- **5.30** (Quality Points for Student's Grades) Write a function toQualityPoints that inputs a student's average and returns 4 if it's 90–100, 3 if it's 80–89, 2 if it's 70–79, 1 if it's 60–69, and 0 if the average is lower than 60.
- **5.31** (Coin Tossing) Write a program that simulates coin tossing. For each toss, display Heads or Tails. Let the program toss the coin 100 times, and count the number of heads and tails. Display the results. The program should call a function flip that takes no arguments and returns 0 for tails and 1 for heads. If the program realistically

simulates the coin tossing, then each side of the coin should appear approximately half the time for a total of approximately 50 heads and 50 tails.

5.32 (Guess the Number) Write a C program that plays the game of "guess the number" as follows: Your program chooses the number to be guessed by selecting an integer at random in the range 1 to 1000. The program then types:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
```

The player types a first guess. The program responds with one of the following:

```
    Excellent! You guessed the number!
    Would you like to play again (y or n)?
    Too low. Try again.
    Too high. Try again.
```

If the guess is incorrect, your program should loop until the player guesses the number. Your program should keep telling the player Too high or Too low to help the player "zero in" on the correct answer.

- **5.33** (Guess the Number Modification) Modify your Exercise 5.32 solution to count the number of guesses the player makes. If the number is 10 or fewer, print "Either you know the secret or you got lucky!" If the player guesses the number in 10 tries, then print "Aha! You know the secret!" If the player makes more than 10 guesses, then print "You should be able to do better!" Why should it take no more than 10 guesses? Well, with each "good guess" the player should be able to eliminate half of the numbers. Now show why any number 1 to 1000 can be guessed in 10 or fewer tries.
- **5.34** (*Recursive Exponentiation*) Write a recursive function power(base, exponent) that when invoked returns

```
baseexponent
```

For example, power (3, 4) = 3 * 3 * 3 * 3. Assume that exponent is an integer greater than or equal to 1. *Hint:* The recursion step would use the relationship

```
base exponent = base * base exponent-1
```

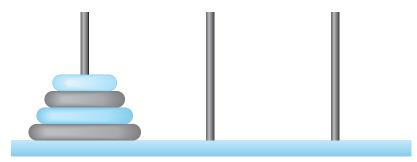
and the terminating condition occurs when exponent is equal to 1 because

```
base^1 = base
```

5.35 *(Fibonacci)* The Fibonacci series

begins with the terms 0 and 1 and has the property that each succeeding term is the sum of the two preceding terms. First, write a *nonrecursive* function fibonacci(n) that calculates the nth Fibonacci number. Use int for the function's parameter and unsigned long long int for its return type. Then, determine the largest Fibonacci number that can be printed on your system.

5.36 (*Towers of Hanoi*) Every budding computer scientist must grapple with certain classic problems, and the Towers of Hanoi (shown in the following diagram) is one of the most famous of these:



Legend has it that in a temple in the Far East, priests are attempting to move a stack of disks from one peg to another. The initial stack had 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from this peg to a second peg under the constraints that exactly one disk is moved at a time and at no time may a larger disk be placed above a smaller disk. A third peg is available for temporarily holding the disks. Supposedly the world will end when the priests complete their task, so there's little incentive for us to facilitate their efforts.

Let's assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that will print the precise sequence of disk-to-disk peg transfers.

If we were to approach this problem with conventional methods, we'd rapidly find ourselves hopelessly knotted up in managing the disks. Instead, if we attack the problem with recursion in mind, it immediately becomes tractable. Moving n disks can be viewed in terms of moving only n-1 disks (and hence the recursion) as follows:

- a) Move n-1 disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
- b) Move the last disk (the largest) from peg 1 to peg 3.
- c) Move the n-1 disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving n = 1 disk, i.e., the base case. This is accomplished by trivially moving the disk without the need for a temporary holding area.

Write a program to solve the Towers of Hanoi problem. Use a recursive function with four parameters:

- a) The number of disks to be moved.
- b) The peg on which these disks are initially threaded.
- c) The peg to which this stack of disks is to be moved.
- d) The peg to be used as a temporary holding area.

Your program should print the precise instructions it will take to move the disks from the starting peg to the destination peg. For example, to move a stack of three disks from peg 1 to peg 3, your program should print the following series of moves:

```
1 \rightarrow 3 (This means move one disk from peg 1 to peg 3.) 1 \rightarrow 2 3 \rightarrow 2 1 \rightarrow 3 2 \rightarrow 1 2 \rightarrow 3 1 \rightarrow 3
```

- **5.37** (*Towers of Hanoi: Iterative Solution*) Any program that can be implemented recursively can be implemented iteratively, although sometimes with considerably more difficulty and considerably less clarity. Try writing an iterative version of the Towers of Hanoi. If you succeed, compare your iterative version with the recursive version you developed in Exercise 5.36. Investigate issues of performance, clarity and your ability to demonstrate the correctness of the programs.
- **5.38** (Visualizing Recursion) It's interesting to watch recursion "in action." Modify the factorial function of Fig. 5.9 to print its local variable and recursive call parameter. For each recursive call, display the outputs on a separate line and add a level of indentation. Do your utmost to make the outputs clear, interesting and meaningful. Your goal here is to design and implement an output format that helps a person understand recursion better. You may want to add such display capabilities to the many other recursion examples and exercises throughout the text.
- **5.39** (*Recursive Greatest Common Divisor*) The greatest common divisor of integers x and y is the largest integer that evenly divides both x and y. Write a recursive function gcd that returns the greatest common divisor of x and y. The greatest common divisor of x and y is defined recursively as follows: If y is equal to 0, then gcd(x, y) is x; otherwise gcd(x, y) is gcd(y, x % y), where % is the remainder operator.
- **5.40** (*Recursive main*) Can main be called recursively? Write a program containing a function main. Include static local variable count initialized to 1. Postincrement and print the value of count each time main is called. Run your program. What happens?
- **5.41** (*Distance Between Points*) Write a function distance that calculates the distance between two points (x1, y1) and (x2, y2). All numbers and return values should be of type double.
- **5.42** What does the following program do? What happens if you exchange lines 7 and 8?

```
#include <stdio.h>
 I
 2
 3
    int main(void) {
 4
        int c = '\setminus 0'; // variable to hold character input by user
 5
 6
       if ((c = getchar()) != EOF) {
 7
           main();
 8
           printf("%c", c);
 9
       }
10
    }
```

5.43 What does the following program do?

```
#include <stdio.h>
 2
 3
    int mystery(int a, int b); // function prototype
 4
 5
    int main(void) {
 6
       printf("%s", "Enter two positive integers: ");
 7
       int x = 0; // first integer
       int y = 0; // second integer
 8
 9
       scanf("%d%d", &x, &y);
10
П
       printf("The result is %d\n", mystery(x, y));
    }
12
13
    // Parameter b must be a positive integer
15
    // to prevent infinite recursion
16
   int mystery(int a, int b) {
17
       // base case
18
       if (1 == b) {
19
          return a;
20
       else { // recursive step
21
22
          return a + mystery(a, b - 1);
23
       }
   }
24
```

- **5.44** After you determine what the program of Exercise 5.43 does, modify it to function properly after removing the restriction that the second argument must be positive.
- **5.45** (*Testing Math Library Functions*) Write a program that tests the math library functions shown in Section 5.3's table. Exercise each of these functions by having your program print out tables of return values for a diversity of argument values.
- **5.46** Find the error in each of the following program segments and explain how to correct it:

```
a) double cube(float); // function prototype
  cube(float number) { // function definition
      return number * number * number;
  }
b) int randomNumber = srand();
c) double y = 123.45678;
  int x;
  x = y;
  printf("%f\n", (double) x);
d) double square(double number) {
      double number;
      return number * number;
}
```

```
e) int sum(int n) {
    if (0 == n) {
        return 0;
    }
    else {
        return n + sum(n);
    }
}
```

- **5.47** (Craps Game Modification) Modify the craps program of Fig. 5.7 to allow wagering. Package as a function the portion of the program that runs one game of craps. Initialize variable bankBalance to 1000 dollars. Prompt the player to enter a wager. Use a while loop to check that wager is less than or equal to bankBalance, and if not, prompt the user to reenter wager until a valid wager is entered. After a correct wager is entered, run one game of craps. If the player wins, increase bankBalance by wager and print the new bankBalance. If the player loses, decrease bankBalance by wager, print the new bankBalance, check whether bankBalance has become zero, and if so print the message, "Sorry. You busted!" As the game progresses, print various messages to create some "chatter" such as, "Oh, you're going for broke, huh?" or "Aw cmon, take a chance!" or "You're up big. Now's the time to cash in your chips!"
- **5.48** (Research Project: Improving the Recursive Fibonacci Implementation) In Section 5.15, the recursive algorithm we used to calculate Fibonacci numbers was intuitively appealing. However, recall that the algorithm resulted in the exponential explosion of recursive function calls. Research the recursive Fibonacci implementation online. Study the various approaches, including the iterative version in Exercise 5.35 and versions that use only so-called "tail recursion." Discuss the relative merits of each.

Computer-Assisted Instruction

Computers create exciting possibilities for improving the educational experience of all students worldwide, as suggested by the next five exercises. [*Note:* Check out initiatives such as the One Laptop Per Child Project (www.laptop.org).]

5.49 (Computer-Assisted Instruction) The use of computers in education is referred to as computer-assisted instruction (CAI). Write a program that will help an elementary-school student learn multiplication. Use the rand function to produce two positive one-digit integers. The program should then prompt the user with a question, such as

```
How much is 6 times 7?
```

The student then inputs the answer. Next, the program checks the student's answer. If it's correct, display the message "Very good!" and ask another multiplication question. If the answer is wrong, display the message "No. Please try again." and let the student try the same question repeatedly until the student finally gets it right. A separate function should be used to generate each new question. This function should be called once when the application begins execution and each time the user answers the question correctly.

5.50 (Computer-Assisted Instruction: Reducing Student Fatigue) One problem in CAI environments is student fatigue. This can be reduced by varying the computer's responses to hold the student's attention. Modify the program of Exercise 5.49 so that various comments are displayed for each answer as follows:

Possible responses to a correct answer:

```
Very good!
Excellent!
Nice work!
Keep up the good work!
```

Possible responses to an incorrect answer:

```
No. Please try again.
Wrong. Try once more.
Don't give up!
No. Keep trying.
```

Use random-number generation to choose a number from 1 to 4 that will be used to select one of the four appropriate responses to each correct or incorrect answer. Use a switch statement to issue the responses.

- **5.51** (Computer-Assisted Instruction: Monitoring Student Performance) More sophisticated computer-assisted instruction systems monitor the student's performance over a period of time. The decision to begin a new topic is often based on the student's success with previous topics. Modify the program of Exercise 5.50 to count the number of correct and incorrect responses typed by the student. After the student types 10 answers, your program should calculate the percentage that are correct. If the percentage is lower than 75%, display "Please ask your teacher for extra help.", then reset the program so another student can try it. If the percentage is 75% or higher, display "Congratulations, you are ready to go to the next level!", then reset the program so another student can try it.
- **5.52** (Computer-Assisted Instruction: Difficulty Levels) Exercises 5.49–5.51 developed a computer-assisted instruction program to help teach an elementary-school student multiplication. Modify the program to allow the user to enter a difficulty level. At a difficulty level of 1, the program should use only single-digit numbers in the problems; at a difficulty level of 2, numbers as large as two digits, and so on.
- **5.53** (Computer-Assisted Instruction: Varying the Types of Problems) Modify the program of Exercise 5.52 to allow the user to pick a type of arithmetic problem to study. An option of 1 means addition problems only, 2 means subtraction problems only, 3 means multiplication problems only and 4 means a random mixture of all these types.

Random-Number Simulation Case Study: The Tortoise and the Hare

5.54 (*The Tortoise and the Hare Race*) In this problem, you'll recreate one of the truly great moments in history—the classic race of the tortoise and the hare. You'll use random-number generation to develop a simulation of this memorable event.

Our contenders begin the race at "square 1" of 70 squares. Each square represents a possible position along the racecourse. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up a slippery mountainside, so occasionally, the contenders lose ground.

There's a clock that ticks once per second. With each tick, adjust the animals' positions according to the following rules:

Animal	Move type	Percentage of the time	Actual move
Tortoise	Fast plod	50%	3 squares forward
	Slip	20%	6 squares backward
	Slow plod	30%	1 square forward
Hare	Sleep	20%	No move at all
	Big hop	20%	9 squares forward
	Big slip	10%	12 squares backward
	Small hop	30%	1 square forward
	Small slip	20%	2 squares backward

Use variables to keep track of the animals' positions (i.e., position numbers are 1–70). Start each animal at position 1 (i.e., the "starting gate"). If an animal slips left before square 1, move the animal back to square 1. If an animal moves past square 70, move the animal back to square 80.

Generate the percentages in the preceding table by producing a random integer, x, in the range $1 \le x \le 10$. For the tortoise, perform a "fast plod" when $1 \le x \le 5$, a "slip" when $6 \le x \le 7$, or a "slow plod" when $8 \le x \le 10$. Use a similar technique to move the hare.

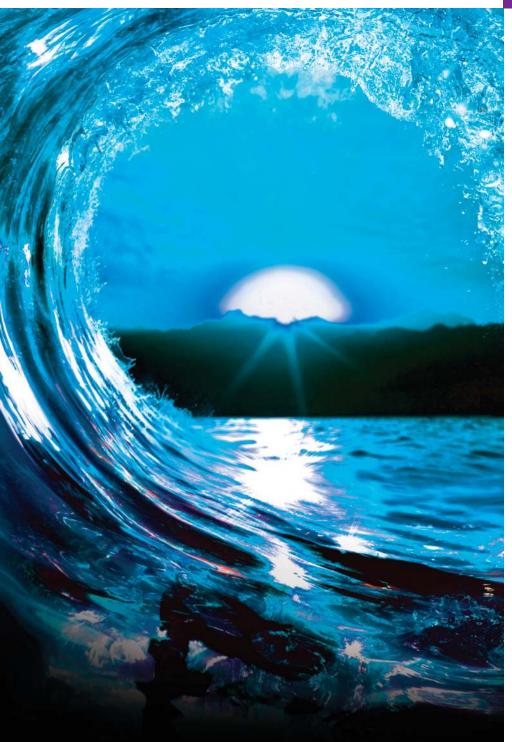
Begin the race by printing

```
ON YOUR MARK, GET SET
BANG !!!!
AND THEY'RE OFF !!!!
```

Then, for each tick (i.e., each iteration of a loop), print a 70-position line showing the letter T in the tortoise's position and the letter H in the hare's position. Occasionally, the contenders will land on the same square. In this case, the tortoise bites the hare, and your program should print "OUCH!!!" beginning at that position. All print positions other than the T, the H, or the OUCH!!! (in case of a tie) should be blank.

After printing each line, test whether either animal has reached or passed square 70. If so, then print the winner and terminate the simulation. If the tortoise wins, print "TORTOISE WINS!!! YAY!!!" If the hare wins, print "Hare wins. Yuch." If both animals win on the same tick of the clock, you may want to favor the turtle (the "underdog"), or you may want to print "It's a tie". If neither animal wins, perform the loop again to simulate the next tick of the clock. When you're ready to run your program, assemble a group of fans to watch the race. You'll be amazed at how involved your audience gets!

Arrays



Objectives

In this chapter, you'll:

- Use the array data structure to represent lists and tables of values.
- Define arrays, initialize arrays and refer to individual elements of arrays.
- Define symbolic constants.
- Pass arrays to functions.
- Use arrays to store, sort and search lists and tables of values.
- Be introduced to data science using basic descriptive statistics, such as mean, median and mode.
- Define and manipulate multidimensional arrays.
- Create variable-length arrays whose size is determined at execution time.
- Understand security issues related to input with scanf, output with printf and arrays.

- **6.1** Introduction
- **6.2** Arrays
- **6.3** Defining Arrays
- **6.4** Array Examples
 - 6.4.1 Defining an Array and Using a Loop to Set the Array's Element Values
 - 6.4.2 Initializing an Array in a Definition with an Initializer List
 - 6.4.3 Specifying an Array's Size with a Symbolic Constant and Initializing Array Elements with Calculations
 - 6.4.4 Summing the Elements of an Array
 - 6.4.5 Using Arrays to Summarize Survey
 - 6.4.6 Graphing Array Element Values with Bar Charts
 - 6.4.7 Rolling a Die 60,000,000 Times and Summarizing the Results in an Array
- **6.5** Using Character Arrays to Store and Manipulate Strings
 - 6.5.1 Initializing a Character Array with a String
 - 6.5.2 Initializing a Character Array with an Initializer List of Characters
 - 6.5.3 Accessing the Characters in a String
 - 6.5.4 Inputting into a Character Array

- 6.5.5 Outputting a Character Array That Represents a String
- 6.5.6 Demonstrating Character Arrays
- **6.6** Static Local Arrays and Automatic Local Arrays
- **6.7** Passing Arrays to Functions
- **6.8** Sorting Arrays
- **6.9** Intro to Data Science Case Study: Survey Data Analysis
- **6.10** Searching Arrays
 - 6.10.1 Searching an Array with Linear Search
 - 6.10.2 Searching an Array with Binary Search
- **6.11** Multidimensional Arrays
 - 6.11.1 Illustrating a Two-Dimensional Array
 - 6.11.2 Initializing a Double-Subscripted Array
 - 6.11.3 Setting the Elements in One Row
 - 6.11.4 Totaling the Elements in a Two-Dimensional Array
 - 6.11.5 Two-Dimensional Array Manipulations
- **6.12** Variable-Length Arrays
- **6.13** Secure C Programming

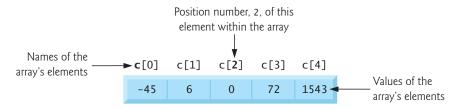
Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises | Recursion Exercises

6.1 Introduction

This chapter introduces data structures. Arrays are data structures consisting of related data items of the same type. Chapter 10 discusses C's notion of struct—a data structure consisting of related data items of possibly different types. Arrays and structs are "static" entities in that they remain the same size throughout their lifetimes.

6.2 Arrays

An array is a group of elements of the same type stored contiguously in memory. The following diagram shows an integer array called c, containing five elements:



To refer to a particular location or element in the array, we specify the array's name, followed by the element's **position number** in square brackets ([]). The first element is located at position number 0 (zero). The position number is called the element's **subscript** (or **index**). A subscript must be a non-negative integer or integer expression.

Let's examine the array in the previous diagram more closely. The array's name is c. The value of c[0] is -45, c[2] is 0 and c[4] is 1543. A subscripted array name is an *lvalue* that can be used on the left side of an assignment. So, the statement:

```
c[2] = 1000;
```

replaces c[2]'s current value (0) with the value 1000. To print the sum of the values in array c's first three elements, we'd write:

```
printf("%d", c[0] + c[1] + c[2]);
```

To divide the value of element 3 of array c by 2 and assign the result to the variable x, write:

```
x = c[3] / 2;
```

The brackets that enclose an array's subscript are an operator with the highest level of precedence. The following table shows the precedence and grouping of the operators introduced to this point in the text:

Operators	Grouping	Туре
[] () ++ (postfix) (postfix)	left to right	highest
+ - ! ++ (prefix) (prefix) (type)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
H	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Self Check

- [(Multiple Choice) Which of the following statements is false?
 - Any array element may be referred to by giving the array's name followed by the element's position number in square brackets ([]).
 - b) Every array's first element has position number 1.
 - c) An array name, like other identifiers, can contain only letters, digits and underscores and cannot begin with a digit.

d) The position number in square brackets is called the element's subscript (or index), which must be an integer or an integer expression.

Answer: b) is false. Actually, every array's first element has position number 0.

2 (*Code*) Write a statement that displays the int product of the values contained in the first four elements of int array grades.

```
Answer: printf("%d", grades[0] * grades[1] * grades[2] * grades[3]);
```

6.3 Defining Arrays

When you define an array, you specify its element type and number of elements so the compiler may reserve the appropriate amount of memory. The following definition reserves five elements for integer array c, which has subscripts in the range 0–4.

```
int c[5];
The definitions
  int b[100];
  int x[27];
```

reserve 100 elements for integer array b and 27 elements for integer array x. These arrays have subscripts in the ranges 0–99 and 0–26, respectively.

A char array can store a character string. Character strings and their similarity to arrays are discussed in Chapter 8. The relationship between pointers and arrays is discussed in Chapter 7.

Self Check

- [(Multiple Choice) Which of the following statements is false?
 - a) When creating an array, you specify an array's element type and number of elements so the compiler may reserve the appropriate amount of memory.
 - b) The following definition reserves space for the double array temperatures, which has index numbers in the range 0–6:

```
double temperatures[7];
```

c) The following definitions reserve 50 elements for float array b and 19 elements for float array x:

```
float b[50];
float x[19];
```

d) An array of type string can store a character string.

Answer: d) is *false*. Actually, an array of type char can store a character string. C does not have a string type.

6.4 Array Examples

This section presents several examples demonstrating how to define and initialize arrays and how to perform many common array manipulations.

6.4.1 Defining an Array and Using a Loop to Set the Array's Element Values

Like any other local variable, uninitialized array elements contain "garbage" values. Figure 6.1 uses for statements to set five-element integer array n's elements to zeros (lines 10–12) and print the array in tabular format (lines 17–19). The first printf statement (line 14) displays the column heads for the two columns printed in the subsequent for statement.

```
// fig06_01.c
I
   // Initializing the elements of an array to zeros.
3
  #include <stdio.h>
4
5
    // function main begins program execution
    int main(void) {
7
       int n[5]; // n is an array of five integers
       // set elements of array n to 0
9
10
       for (size_t i = 0; i < 5; ++i) {
          n[i] = 0; // set element at location i to 0
11
12
13
       printf("%s%8s\n", "Element", "Value");
14
15
       // output contents of array n in tabular format
16
17
       for (size_t i = 0; i < 5; ++i) {
          printf("%7zu%8d\n", i, n[i]);
18
19
    }
```

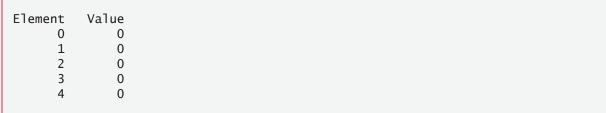


Fig. 6.1 | Initializing the elements of an array to zeros.

The counter-control variable i's type is size_t in each for statement (lines 10 and 17). The C standard says size_t represents an unsigned integral type and is recommended for any variable representing an array's size or subscripts. Type size_t is defined in header <stddef.h>, which is often included by other headers (such as <stdio.h>).¹ The conversion specification %zu is used to display size_t values.

^{1.} If you attempt to compile Fig. 6.1 and receive errors, include <stddef.h> in your program.

6.4.2 Initializing an Array in a Definition with an Initializer List

You can initialize an array's elements when defining the array by providing a commaseparated list of array initializers in braces, {}. Figure 6.2 initializes an integer array with five values (line 7) and prints it in a tabular format.

```
I // fig06_02.c
2 // Initializing the elements of an array with an initializer list.
3 #include <stdio.h>
4
5 // function main begins program execution
   int main(void) {
7
       int n[5] = {32, 27, 64, 18, 95}; // initialize n with initializer list
8
       printf("%s%8s\n", "Element", "Value");
9
10
11
       // output contents of array in tabular format
12
       for (size_t i = 0; i < 5; ++i) {
          printf("%7zu%8d\n", i, n[i]);
13
14
       }
15
   }
          Value
Element
             32
      0
      1
             27
      2
```

Fig. 6.2 Initializing the elements of an array with an initializer list.

64

18 95

3

If there are fewer initializers than array elements, the remaining elements are initialized to 0. For example, Fig. 6.1 could have initialized array n's elements to zero as follows:

```
int n[5] = \{0\}; // initializes entire array to zeros
```

This explicitly initializes n[0] to 0 and implicitly initializes the remaining elements to ERR (X) 0. It's a compilation error if you provide more initializers in an array initializer list than elements in the array. For example, the following array definition produces a compilation error because there are four initializers for only three elements:

```
int n[3] = \{32, 27, 64, 18\};
```

The following definition creates a five-element array initialized with the values 1–5: int $n[] = \{1, 2, 3, 4, 5\};$

When you omit the array size, the compiler calculates the array's number of elements from the number initializers.

6.4.3 Specifying an Array's Size with a Symbolic Constant and Initializing Array Elements with Calculations

Figure 6.3 initializes five-element array s with the values 2, 4, 6, 8 and 10, then prints the array in tabular format. To generate the values, we multiply the loop counter by 2 and add 2.

```
// fig06_03.c
 1
 2 // Initializing the elements of array s to the even integers from 2 to 10.
 3 #include <stdio.h>
   #define SIZE 5 // maximum size of array
 4
 5
    // function main begins program execution
 6
 7
    int main(void) {
       // symbolic constant SIZE can be used to specify array size
 8
 9
       int s[SIZE] = {0}; // array s has SIZE elements
10
       for (size_t j = 0; j < SIZE; ++j) { // set the values}
11
12
          s[j] = 2 + 2 * j;
13
       }
14
       printf("%s%8s\n", "Element", "Value");
15
16
17
       // output contents of array s in tabular format
18
       for (size_t j = 0; j < SIZE; ++j) {
          printf("%7zu%8d\n", j, s[j]);
19
20
       }
21
    }
Element
          Value
```

```
Element Value
0 2
1 4
2 6
3 8
4 10
```

Fig. 6.3 Initializing the elements of array s to the even integers from 2 to 10.

Line 4 uses the #define preprocessor directive

```
#define SIZE 5
```

to create the symbolic constant SIZE with the value 5. A symbolic constant is an identifier that the C preprocessor replaces with replacement text before the program is compiled. In this program, the preprocessor replaces all SIZE occurrences with 5.

Using symbolic constants to specify array sizes makes programs easier to read and modify. In Fig. 6.3, for example, we could have the first for loop (line 11) fill a 1000-element array simply by changing SIZE's value in the #define directive from 5 to 1000. Without the symbolic constant, we'd have to change the program in lines 9, 11 and 18. As programs get larger, this technique becomes more useful for writing clear, easy-to-read, maintainable programs. A symbolic constant (like SIZE) is easier to understand than the numeric value 5, which could have different meanings throughout the code.

Do not terminate the #define preprocessor directives with semicolons. If you do that in line 4, the preprocessor replaces all occurrences of SIZE with the text "5;".

ERR

This may lead to syntax errors at compile time or logic errors at execution time. Remember that the preprocessor is not the C compiler.

ERR 🛞

Assigning a value to a symbolic constant in an executable statement is a compilation error—symbolic constants are not variables. By convention, use only uppercase letters for symbolic constant names, so they stand out in a program. This also reminds you that symbolic constants are not variables.

6.4.4 Summing the Elements of an Array

Figure 6.4 sums the values contained in the five-element integer array a. The for statement's body (line 14) does the totaling.

```
I // fig06_04.c
2
   // Computing the sum of the elements of an array.
3 #include <stdio.h>
4 #define SIZE 5
5
   // function main begins program execution
6
7
   int main(void) {
       // use an initializer list to initialize the array
8
9
       int a[SIZE] = \{1, 2, 3, 4, 5\};
10
       int total = 0; // sum of array
П
       // sum contents of array a
12
       for (size_t i = 0; i < SIZE; ++i) {
13
14
          total += a[i];
15
16
       printf("The total of a's values is %d\n", total);
17
18
    }
The total of a's values is 15
```

Fig. 6.4 Computing the sum of the elements of an array.

6.4.5 Using Arrays to Summarize Survey Results

Our next example uses arrays to summarize the results of data collected in a survey. Consider the problem statement:

Twenty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 5 (1 means awful, and 5 means excellent). Place the 20 responses in an integer array and summarize the results of the poll.

Figure 6.5 is a typical array application. We wish to summarize the number of responses of each type. The 20-element array responses (lines 10–11) contains the students' responses. We use a 6-element array frequency (line 14) to count each response's number of occurrences. We ignore frequency[0] because it's logical to

have response 1 increment frequency[1] rather than frequency[0]. This allows us to use each response directly as the subscript in the frequency array. You should strive for program clarity. Sometimes it may be worthwhile to trade off the most efficient >> PERF use of memory or processor time in favor of writing clearer programs. Sometimes performance considerations far outweigh clarity considerations.

```
I // fig06_05.c
 2 // Analyzing a student poll.
 3 #include <stdio.h>
 4 #define RESPONSES_SIZE 20 // define array sizes
 5 #define FREQUENCY_SIZE 6
 7 // function main begins program execution
   int main(void) {
 8
 9
       // place the survey responses in the responses array
10
       int responses[RESPONSES_SIZE] =
          \{1, 2, 5, 4, 3, 5, 2, 1, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 2, 5\};
П
12
13
       // initialize frequency counters to 0
14
       int frequency[FREQUENCY_SIZE] = {0};
15
16
       // for each answer, select the value of an element of the array
17
       // responses and use that value as a subscript into the array
18
       // frequency to determine the element to increment
       for (size_t answer = 0; answer < RESPONSES_SIZE; ++answer) {</pre>
19
20
          ++frequency[responses[answer]];
21
       }
22
23
       // display results
       printf("%s%12s\n", "Rating", "Frequency");
24
25
       // output the frequencies in a tabular format
26
       for (size_t rating = 1; rating < FREQUENCY_SIZE; ++rating) {</pre>
27
          printf("%6zu%12d\n", rating, frequency[rating]);
28
29
30
    }
Rating
         Frequency
     1
                  3
                  5
     2
     3
                  7
     4
                  2
     5
                  3
```

Fig. 6.5 Analyzing a student poll.

How the frequency Counters Are Incremented

The for loop (lines 19–21) takes each response from responses and increments one of the five frequency array counters—frequency[1] to frequency[5]. The key statement in the loop is line 20:

```
++frequency[responses[answer]];
```

which increments the appropriate frequency counter, based on the value of the expression responses [answer]. When the counter variable answer is 0, responses [answer] is 1, so ++frequency[responses[answer]]; is interpreted as

```
++frequency[1];
```

which increments frequency[1]. When answer is 1, the value of responses[answer] is 2, so ++frequency[responses[answer]]; is interpreted as

```
++frequency[2];
```

which increments frequency[2]. When answer is 2, the value of responses[answer] is 5, so ++frequency[responses[answer]]; is interpreted as

```
++frequency[5];
```

which increments frequency [5], and so on.

Invalid Survey Responses

Regardless of the number of survey responses processed, only a 6-element frequency array is required (ignoring element zero) to summarize the results. But what if the data contained an invalid value such as 13? In this case, the program would attempt to add 1 to frequency[13], which is outside the array's bounds. C has no array bounds checking to prevent a program from referring to an element that does not exist. So, an exe-SEC cuting program can "walk off" either end of an array without warning—a security problem we discuss in Section 6.13. Programs should validate that all input values are correct to prevent erroneous information from affecting a program's calculations.

Validate Array Subscripts



ERR (X) Referring to an element outside the array bounds is a logic error. When looping through an array, the array subscript should never go below 0 and should always be less than the total number of array elements—the array's size minus one. You should ensure that all array references remain within the bounds of the array.

6.4.6 Graphing Array Element Values with Bar Charts

Our next example (Fig. 6.6) reads numbers from an array and graphs the information in a bar chart. We display each number followed by a bar consisting of that many asterisks. The nested for statement (lines 17–19) displays the bars by iterating n[i] times and displaying one asterisk per iteration. Line 21 ends each bar.

```
// fig06_06.c
2 // Displaying a bar chart.
3 #include <stdio.h>
4 #define SIZE 5
  // function main begins program execution
7 int main(void) {
```

Displaying a bar chart. (Part 1 of 2.)

```
// use initializer list to initialize array n
 8
       int n[SIZE] = \{19, 3, 15, 7, 11\};
 9
10
       printf("%s%13s%17s\n", "Element", "Value", "Bar Chart");
П
12
       // for each element of array n, output a bar of the bar chart
13
       for (size_t i = 0; i < SIZE; ++i) {
14
          printf("%7zu%13d%8s", i, n[i], "");
15
16
           for (int j = 1; j \leftarrow n[i]; ++j) { // print one bar
17
             printf("%c", '*');
18
19
20
21
          puts(""); // end a bar with a newline
22
       }
23
   }
```

Value	Bar Chart
19	*********
3	***
15	*******
7	*****
11	*****

Fig. 6.6 Displaying a bar chart. (Part 2 of 2.)

6.4.7 Rolling a Die 60,000,000 Times and Summarizing the Results in an Array

In Chapter 5, we stated that we'd show a more elegant way to write Fig. 5.5's dicerolling program. Recall that the program rolled a single six-sided die 60,000,000 times and displayed the face counts. Figure 6.7 is an array version of Fig. 5.5. Line 17 replaces Fig. 5.5's entire 20-line switch statement. Once again, we use a frequency array in which we ignore element 0, so we can use the face values as subscripts into the array.

```
I // fig06_07.c
2 // Roll a six-sided die 60,000,000 times
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <time.h>
6
   #define SIZE 7
7
   // function main begins program execution
8
9
    int main(void) {
       srand(time(NULL)); // seed random number generator
10
П
       int frequency[SIZE] = {0}; // initialize all frequency counts to 0
12
13
```

Fig. 6.7 | Roll a six-sided die 60,000,000 times. (Part 1 of 2.)

```
// roll die 60,000,000 times
14
15
       for (int roll = 1; roll <= 60000000; ++roll) {
16
          size_t face = 1 + rand() \% 6;
17
          ++frequency[face]; // replaces entire switch of Fig. 5.5
       }
18
19
       printf("%s%17s\n", "Face", "Frequency");
20
21
       // output frequency elements 1-6 in tabular format
22
23
       for (size_t face = 1; face < SIZE; ++face) {</pre>
          printf("%4zu%17d\n", face, frequency[face]);
24
25
       }
26 }
```

```
Face Frequency
1 9997167
2 10003506
3 10001940
4 9995833
5 10000843
6 10000711
```

Fig. 6.7 | Roll a six-sided die 60,000,000 times. (Part 2 of 2.)

Self Check

(Code) Rewrite the following code segment to define seven-element double array m and initialize each of its elements to 10:

```
// set elements of array n to 0
for (size_t i = 0; i < 5; ++i) {
    n[i] = 0; // set element at location i to 0
}

Answer:
    double m[7]; // m is an array of 7 doubles

    // set elements of array m to 10.0
    for (size_t i = 0; i < 7; ++i) {
        m[i] = 10.0; // set element at location i to 10.0
}</pre>
```

int n[5]; // n is an array of five integers

- **2** (Multiple Choice) Which of the following statements a), b) or c) is true?
 - a) For an int array, if you provide fewer initializers than there are elements in the array, the remaining elements are initialized to 0.
 - b) It's a syntax error to provide more initializers in an array initializer list than there are array elements—for example, int n[3] = {32, 27, 64, 18}; is a syntax error, because there are four initializers but only three array elements.

c) If the array size is omitted from a definition with an initializer list, the compiler determines the number of elements based on the number of elements in the initializer list. So, the following creates the three-element int array s:

```
int s[] = \{10, 20, 30\};
```

d) All of the above statements are *true*.

Answer: d.

6.5 Using Character Arrays to Store and Manipulate Strings

Arrays can hold data of any type, though all the elements of a given array must have the same type. We now discuss storing strings in character arrays. So far, the only string-processing capability we have is outputting a string with printf. A string such as "hello" is really an array of individual characters, plus one more thing.

6.5.1 Initializing a Character Array with a String

Character arrays have several unique features. A character array can be initialized using a string literal. For example,

```
char string1[] = "first";
```

initializes the elements of array string1 to the individual characters in the string literal "first". In this case, the compiler determined array string1's size based on the string's length. The string "first" contains five characters plus a string-terminating null character. So, string1 actually contains six elements. The escape sequence representing the null character is '\0'. All strings end with this character. A character array representing a string should always be defined large enough to hold the string's number of characters and the terminating null character.

6.5.2 Initializing a Character Array with an Initializer List of Characters

Character arrays also can be initialized with individual character constants in an initializer list, but this can be tedious. The preceding definition is equivalent to

```
char string1[] = {'f', 'i', 'r', 's', 't', '\0'};
```

6.5.3 Accessing the Characters in a String

You can access a string's individual characters directly using array subscript notation. So, string1[0] is the character 'f', string1[3] is 's' and string1[5] is '\0'.

6.5.4 Inputting into a Character Array

The following definition creates a character array capable of storing a string of *at most* 19 characters and a terminating null character:

```
char string2[20];
```

SEC 🕤

The statement

```
scanf("%19s", string2);
```

reads a string from the keyboard into string2. You pass the array name to scanf without the & used with non-string variables. The & is normally used to provide scanf with a variable's *location* in memory so that a value can be stored there. In Section 6.7, when we discuss passing arrays to functions, we'll discuss why the & is not necessary for array names.

It's your responsibility to ensure that the array into which the string is read is capable of holding any string that the user types at the keyboard. Function scanf does *not* check how large the array is. It will read characters until a space, tab, newline or end-offile indicator is encountered. The string string2 should be no longer than 19 characters to leave room for the terminating null character. If the user types 20 or more characters, your program may crash or create a security vulnerability called a buffer overflow. For this reason, we used the conversion specification %19s. This tells scanf to read a maximum of 19 characters, preventing it from writing characters into memory beyond the end of string2. (In Section 6.13, we revisit the potential security issue raised by inputting into a character array and discuss the C standard's scanf_s function.)

6.5.5 Outputting a Character Array That Represents a String

A character array representing a string can be output with printf using the %s conversion specification. For example, you can print the character array string2 with

```
printf("%s\n", string2);
```

Like scanf, printf does not check how large the character array is. It displays the string's characters until it encounters a terminating null character. [Consider what would print if, for some reason, the terminating null character were missing.]

6.5.6 Demonstrating Character Arrays

Figure 6.8 demonstrates initializing a character array with a string literal, reading a string into a character array, printing a character array as a string and accessing a string's individual characters. The program uses a for statement (lines 20–22) to loop through the string1 array and print the individual characters separated by spaces, using the %c conversion specification. The condition in the for statement is true while the counter is less than the array's size and the terminating null character has not been encountered in the string. This program reads only strings that do not contain whitespace characters. We'll show how to read strings containing whitespace characters in Chapter 8.

```
I // fig06_08.c
```

^{2 //} Treating character arrays as strings.

^{3 #}include <stdio.h>

[#]define SIZE 20

```
5
   // function main begins program execution
   int main(void) {
 7
       char string1[SIZE] = ""; // reserves 20 characters
 8
       char string2[] = "string literal"; // reserves 15 characters
 9
10
       // prompt for string from user then read it into array string1
П
       printf("%s", "Enter a string (no longer than 19 characters): ");
12
       scanf("%19s", string1); // input no more than 19 characters
13
14
       // output strings
15
       printf("string1 is: %s\nstring2 is: %s\n", string1, string2);
16
       puts("string1 with spaces between characters is:");
17
18
19
       // output characters until null character is reached
       for (size_t i = 0; i < SIZE && string1[i] != '\0'; ++i) {
20
          printf("%c ", string1[i]);
21
22
23
24
       puts("");
25 }
Enter a string (no longer than 19 characters): Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
He 11 o
```

Fig. 6.8 | Treating character arrays as strings. (Part 2 of 2.)

Self Check

- [Multiple Choice] Which of the following statements a), b) or c) is false?
 - a) A character array representing a string can be output with printf and the %s conversion specifier, as in:

```
printf("%s\n", month);
```

- b) Function printf, like scanf, does not check how large the character array is.
- c) When function printf displays the characters of a character array representing a string, it stops when it tries to print the first character past the end of the array.
- d) All of the above statements are true.

Answer: c) is *false*. Actually, printf keeps displaying characters until it encounters a terminating null character, even if that is well beyond the end of the array.

2 (*TruelFalse*) The following array can store a string of at most 20 characters and a terminating null character:

```
char name1[20];
```

Answer: False. Actually, the statement creates a character array capable of storing a string of at most 19 characters and a terminating null character.

6.6 Static Local Arrays and Automatic Local Arrays

Chapter 5 discussed the storage-class specifier static. A static local variable exists for the program's duration but is visible only in the function body. We can apply static to a local array definition to prevent the array from being created and initialized every time the function is called and destroyed every time the function exits. This reduces program execution time, particularly for programs with frequently called functions that contain large arrays. Arrays that are static are initialized once at program startup. If you do not explicitly initialize a static array, that array's elements are initialized to zero by default.

Figure 6.9 demonstrates function staticArrayInit (lines 21–38) with a local static array (line 23) and function automaticArrayInit (lines 41–58) with a local automatic array (line 43). Function staticArrayInit is called twice (lines 11 and 15). The local static array in the function is initialized to zero at program startup (line 23). The function prints the array, adds 5 to each element and prints the array again. The second time the function is called, the static array contains the values stored during the first call.

```
// fig06_09.c
 2
    // Static arrays are initialized to zero if not explicitly initialized.
 3
    #include <stdio.h>
 4
 5
    void staticArrayInit(void); // function prototype
 6
    void automaticArrayInit(void); // function prototype
 7
 8
    // function main begins program execution
 9
    int main(void) {
10
       puts("First call to each function:");
11
       staticArrayInit();
12
       automaticArrayInit();
13
       puts("\n\nSecond call to each function:");
14
15
       staticArrayInit();
16
       automaticArrayInit();
17
       puts("");
18
    }
19
20
    // function to demonstrate a static local array
    void staticArrayInit(void) {
21
22
       // initializes elements to 0 before the function is called
23
       static int array1[3];
24
25
       puts("\nValues on entering staticArrayInit:");
26
27
       // output contents of array1
28
       for (size_t i = 0; i <= 2; ++i) {
          printf("array1[%zu] = %d ", i, array1[i]);
29
30
       }
```

Fig. 6.9 Static arrays are initialized to zero if not explicitly initialized. (Part 1 of 2.)

```
31
32
       puts("\nValues on exiting staticArrayInit:");
33
       // modify and output contents of array1
34
       for (size_t i = 0; i <= 2; ++i) {
35
          printf("array1[\%zu] = \%d ", i, array1[i] += 5);
36
37
       }
38
    }
39
40
    // function to demonstrate an automatic local array
41
    void automaticArrayInit(void) {
42
       // initializes elements each time function is called
43
       int array2[3] = \{1, 2, 3\};
44
45
       puts("\n\nValues on entering automaticArrayInit:");
46
47
       // output contents of array2
48
       for (size_t i = 0; i <= 2; ++i) {
49
          printf("array2[%zu] = %d ", i, array2[i]);
50
51
52
       puts("\nValues on exiting automaticArrayInit:");
53
54
       // modify and output contents of array2
       for (size_t i = 0; i <= 2; ++i) {
55
          printf("array2[\%zu] = \%d ", i, array2[i] += 5);
56
57
       }
58
   }
First call to each function:
Values on entering staticArrayInit:
array1[0] = 0 array1[1] = 0 array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5 array1[1] = 5 array1[2] = 5
Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6 \quad array2[1] = 7 \quad array2[2] = 8
Second call to each function:
Values on entering staticArrayInit:
array1[0] = 5 array1[1] = 5 array1[2] = 5 — values preserved from last call
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10
Values on entering automaticArrayInit:
array2[0] = 1 array2[1] = 2 array2[2] = 3 — values reinitialized after last call
Values on exiting automaticArrayInit:
array2[0] = 6 \quad array2[1] = 7 \quad array2[2] = 8
```

Fig. 6.9 | Static arrays are initialized to zero if not explicitly initialized. (Part 2 of 2.)

Function automaticArrayInit is also called twice (lines 12 and 16). The automatic local array's elements are initialized with the values 1, 2 and 3 (line 43). The function prints the array, adds 5 to each element and prints the array again. The second time the function is called, the array elements are initialized to 1, 2 and 3 again because the array has automatic storage duration

Self Check

- [Multiple Choice] Which of the following statements a), b) or c) is false?
 - a) A static local variable exists for the duration of the program but is visible only in the function body.
 - b) A static local array is created and initialized once, not each time the function is called. This reduces program execution time, particularly for programs with frequently called functions that contain large arrays.
 - c) If you do not explicitly initialize a static array, that array's elements are initialized to zero by default.
 - d) All of the above statements are true.

Answer: d.

6.7 Passing Arrays to Functions

To pass an array argument to a function, specify the array's name without any brackets. For example, if array hourlyTemperatures has been defined as

int hourlyTemperatures[HOURS_IN_A_DAY];

the function call

modifyArray(hourlyTemperatures, HOURS_IN_A_DAY)

passes array hourlyTemperatures and its size to function modifyArray.

Recall that all arguments in C are passed by value. However, C automatically passes arrays to functions by reference—the called functions can modify the callers' original array element values. We'll see in Chapter 7 that this is not a contradiction. An array's name evaluates to the address in memory of the array's first element. Because the array's starting address is passed, the called function knows precisely where the array is stored. So, any modifications the called function makes to the array elements change the values of the original array's elements in the caller.

Showing That an Array Name Is an Address

Figure 6.10 demonstrates that "the value of an array name" is really the *address* of the array's first element by printing array, &array[0] and &array using the *p conversion specification for printing addresses. The *p conversion specification normally outputs addresses as hexadecimal numbers, but this is compiler-dependent. Hexadecimal (base 16) numbers consist of the digits 0 through 9 and the letters A through F—the hexadecimal equivalents of the decimal numbers 10–15. Online Appendix E provides an in-depth discussion of the relationships among binary (base 2), octal (base 8), decimal (base 10; standard integers) and hexadecimal integers. The output shows

that array, & array and & array[0] have the same value. This program's output is system dependent, but the addresses are always identical for each program execution on a particular computer.

```
// fig06_10.c
I
2
   // Array name is the same as the address of the array's first element.
3 #include <stdio.h>
5
   // function main begins program execution
   int main(void) {
7
       char array[5] = ""; // define an array of size 5
8
       printf(" array = \%p \n\&array[0] = \%p \n \&array = \%p \n",
9
         array, &array[0], &array);
10
11
   }
    array = 0031F930
&array[0] = 0031F930
   &array = 0031F930
```

Fig. 6.10 Array name is the same as the address of the array's first element.

Passing arrays by reference makes sense for performance reasons. If arrays were 📝 PERF passed by value, a copy of each element would be passed. For large, frequently passed arrays, this would be time-consuming and would consume storage for the copies of the arrays. It's possible to pass an array by value (by placing it in a struct, as we ASE explain in Chapter 10, Structures, Unions, Bit Manipulation and Enumerations).

Passing Individual Array Elements

Although entire arrays are passed by reference, individual array elements are passed by value, exactly as any other variable. Single pieces of data, such as individual ints, floats and chars, are called scalars. To pass an array element to a function, use the subscripted array name as an argument in the function call. In Chapter 7, we show how to pass scalars (i.e., individual variables and array elements) to functions by reference.

Array Parameters

For a function to receive an array through a function call, the function's parameter list must expect an array. The function header for function modifyArray (from earlier in this section) can be written as

```
void modifyArray(int b[], size_t size)
```

indicating that modifyArray expects to receive an int array in parameter b and the array's number of elements in parameter size. The array's number of elements is not required in the array parameter's brackets. If it's included, the compiler checks that it's greater than zero, then ignores it—specifying a negative value is a compilation (X) ERR error. When the called function uses the array name b, it will be referring to the caller's original array. So, in the function call:



```
modifyArray(hourlyTemperatures, HOURS_IN_A_DAY)
```

modifyArray's parameter b represents hourlyTemperatures in the caller. In Chapter 7, we introduce other notations for indicating that a function receives an array. As we'll see, these notations are based on the intimate relationship between arrays and pointers.

Difference Between Passing an Entire Array and Passing an Array Element

Figure 6.11 demonstrates the difference between passing an entire array and passing an individual array element. The program first prints integer array a's five elements (lines 18–20). Next, we pass array a and its size to modifyArray (line 24), which multiplies each of a's elements by 2 (lines 45–47). Then, lines 28–30 display a's updated contents. As the output shows, modifyArray did, indeed, modify a's elements. Next, line 34 prints a[3]'s value and line 36 passes it to function modifyElement. The function multiplies its argument by 2 (line 53) and prints the new value. When line 39 in main displays a[3] again, it has *not* been modified because individual array elements are passed by value.

```
I // fig06_11.c
 2 // Passing arrays and individual array elements to functions.
 3 #include <stdio.h>
 4 #define SIZE 5
 5
 6 // function prototypes
    void modifyArray(int b[], size_t size);
 7
 8 void modifyElement(int e);
 9
   // function main begins program execution
10
   int main(void) {
11
       int a[SIZE] = \{0, 1, 2, 3, 4\}; // initialize array a
12
13
       puts("Effects of passing entire array by reference:\n\nThe "
14
15
          "values of the original array are:");
16
       // output original array
17
       for (size_t i = 0; i < SIZE; ++i) {
18
19
          printf("%3d", a[i]);
20
21
       puts(""); // outputs a newline
22
23
       modifyArray(a, SIZE); // pass array a to modifyArray by reference
24
       puts("The values of the modified array are:");
25
26
27
       // output modified array
       for (size_t i = 0; i < SIZE; ++i) {
28
29
          printf("%3d", a[i]);
       }
30
31
```

Fig. 6.11 | Passing arrays and individual array elements to functions. (Part 1 of 2.)

```
32
       // output value of a[3]
       printf("\n\nEffects of passing array element "
33
          "by value:\n\n value of a[3] is %d\n", a[3]);
34
35
36
       modifyElement(a[3]); // pass array element a[3] by value
37
38
       // output value of a[3]
39
       printf("The value of a[3] is %d\n", a[3]);
40
   }
41
   // in function modifyArray, "b" points to the original array "a" in memory
42
43 void modifyArray(int b[], size_t size) {
44
       // multiply each array element by 2
45
       for (size_t j = 0; j < size; ++j) {
46
          b[j] *= 2; // actually modifies original array
47
48
   }
49
   // in function modifyElement, "e" is a local copy of array element
50
51 // a[3] passed from main
52 void modifyElement(int e) {
       e *= 2; // multiply parameter by 2
53
       printf("Value in modifyElement is %d\n", e);
54
55 }
Effects of passing entire array by reference:
The values of the original array are:
  0 1 2 3 4
The values of the modified array are:
  0 2 4 6 8
Effects of passing array element by value:
The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```

Fig. 6.11 | Passing arrays and individual array elements to functions. (Part 2 of 2.)

Using const to Prevent Functions from Modifying Array Elements

There may be situations in your programs in which a function should *not* modify array elements. C's type qualifier **const** (short for "constant") can prevent a function from modifying an argument. When an array parameter is preceded by the const qualifier, the function treats the array elements as constants. Any attempt to modify an array element in the function body results in a compile-time error. This is another example of the principle of least privilege. A function should not be given the capability to modify an array in the caller unless it's absolutely necessary.

The following definition of a function named tryToModifyArray uses the parameter const int b[] (line 3) to specify that array b is constant and cannot be modified:

```
// in function tryToModifyArray, array b is const, so it cannot be
// used to modify its array argument in the caller
void tryToModifyArray(const int b[]) {
   b[0] /= 2; // error
   b[1] /= 2; // error
   b[2] /= 2; // error
}
```

Each of the function's attempts to modify array elements results in a compiler error. The const qualifier is discussed in additional contexts in Chapter 7.

Self Check

- [(Multiple Choice) Which of the following statements is false?
 - a) Given the following array hourlyTemperatures:

```
int hourlyTemperatures[HOURS_IN_A_DAY];
```

the following function call passes hourlyTemperatures and its size to modifyArray:

```
modifyArray(hourlyTemperatures, HOURS_IN_A_DAY)
```

- b) Recall that all arguments in C are passed by value, so C automatically passes arrays to functions by value.
- c) The array's name evaluates to the address of the array's first element.
- d) Because the address of the array's first element is passed, the called function knows precisely where the array is stored.

Answer: b) is *false*. Actually, C automatically passes arrays to functions by reference—the called functions can modify the element values in the callers' original arrays.

2 (*Discussion*) Based on the meaningful function name and parameter definitions in the following function header, describe as much as you can about what this function probably does:

```
void modifyArray(int b[], size_t size)
```

Answer: Function modifyArray expects to receive an array of integers in parameter b and the number of array elements in parameter size. The array is passed in by reference, so the function is able to modify the original array in the caller.

6.8 Sorting Arrays

Sorting data—that is, placing the data into ascending or descending order—is one of the most important computing applications. A bank sorts checks by account number to prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name and, within that, by first name to make it easy to find phone numbers. Virtually every organization must sort some data, and in many cases, massive amounts of it. Sorting data is an intriguing problem that has attracted some of the most intense computer-science research efforts. Here, we dis-

cuss a simple sorting scheme. In Chapters 12 and 13, we investigate more complex schemes that yield better performance. Often, the simplest algorithms perform \Rightarrow PFRF poorly. Their virtue is that they're easy to write, test and debug. More complex algorithms are often needed to realize maximum performance.

Bubble Sort

Figure 6.12 sorts the 10-element array a's values (line 8) into ascending order. The technique we use is called the bubble sort or the sinking sort because the smaller values gradually "bubble" their way to the top of the array like air bubbles rising in water, while the larger values sink to the bottom of the array. The technique uses several passes through the array. On each pass, the algorithm compares successive pairs of elements (element 0 and element 1, then element 1 and element 2, etc.). If a pair is in increasing order (or if the values are identical), we leave the values as they are. If a pair is in decreasing order, we swap their values in the array.

```
// fig06_12.c
 1
   // Sorting an array's values into ascending order.
 3 #include <stdio.h>
 4 #define SIZE 10
 5
 6
    // function main begins program execution
 7
    int main(void) {
       int a[SIZE] = \{2, 6, 4, 8, 10, 12, 89, 68, 45, 37\};
 8
 9
10
       puts("Data items in original order");
П
12
       // output original array
13
       for (size_t i = 0; i < SIZE; ++i) {
14
          printf("%4d", a[i]);
15
       }
16
17
       // bubble sort
       // loop to control number of passes
18
       for (int pass = 1; pass < SIZE; ++pass) {</pre>
19
20
          // loop to control number of comparisons per pass
21
          for (size_t i = 0; i < SIZE - 1; ++i) {
22
             // compare adjacent elements and swap them if first
23
             // element is greater than second element
24
             if (a[i] > a[i + 1]) {
25
                int hold = a[i];
26
                a[i] = a[i + 1];
27
                a[i + 1] = hold;
28
             }
29
          }
30
31
32
       puts("\nData items in ascending order");
33
```

Fig. 6.12 | Sorting an array's values into ascending order. (Part 1 of 2.)

```
34
      // output sorted array
      for (size_t i = 0; i < SIZE; ++i) {
35
         printf("%4d", a[i]);
36
37
38
39
      puts("");
   }
40
Data items in original order
     6 4 8 10 12 89 68 45 37
Data items in ascending order
         6 8 10 12
                        37 45 68 89
```

Fig. 6.12 | Sorting an array's values into ascending order. (Part 2 of 2.)

First, the program compares a[0] to a[1], then a[1] to a[2], then a[2] to a[3], and so on until it completes the pass by comparing a[8] to a[9]. Although there are 10 elements, only nine comparisons are performed. A large value may move down the array many positions on a single pass because of how the successive comparisons are made, but a small value may move up only one position.

On the first pass, the largest value is guaranteed to sink to the array's bottom element, a[9]. On the second pass, the second-largest value is guaranteed to sink to a[8]. On the ninth pass, the ninth-largest value sinks to a[1]. This leaves the smallest value in a[0], so only nine passes are needed to sort the 10-element array.

Swapping Elements

The sorting is performed by the nested for loops (lines 19–30). If a swap is necessary, it's performed by the three assignments in lines 25–27

```
int hold = a[i];
a[i] = a[i + 1];
a[i + 1] = hold;
```

The variable hold temporarily stores one of the two values being swapped. The swap cannot be performed with only the two assignments

```
a[i] = a[i + 1];

a[i + 1] = a[i];
```

If, for example, a[i] is 7 and a[i + 1] is 5, after the first assignment, both values will be 5 and the value 7 will be lost—hence the need for the extra variable hold.

Bubble Sort Is Easy to Implement, But Slow

The chief virtue of the bubble sort is that it's easy to program. However, it runs slowly because every exchange moves an element only one position closer to its final destination. This becomes apparent when sorting large arrays. In the exercises, we'll develop more efficient versions of the bubble sort. Far more efficient sorts than the bubble sort have been developed. We'll investigate other algorithms in Chapter 13. More advanced courses investigate sorting and searching in greater depth.

Self Check

- [(Multiple Choice) Which of the following statements a), b) or c) is true?
 - a) The chief virtue of the bubble sort is that it's easy to program.
 - b) Bubble sort runs slowly because every exchange moves an element only one position closer to its final position. This becomes apparent when sorting large arrays.
 - c) Far more efficient sorts than the bubble sort have been developed.
 - d) All of the above statements are true.

Answer: d.

2 (*True/False*) If a swap is necessary in the bubble sort, it uses the assignments:

```
a[i] = a[i + 1];
a[i + 1] = a[i];
```

Answer: *False.* Actually, the swap requires one more variable and one more statement:

```
int hold = a[i];
a[i] = a[i + 1];
a[i + 1] = hold;
```

where the variable hold temporarily stores one of the two values being swapped.

6.9 Intro to Data Science Case Study: Survey Data Analysis

We now consider a larger example. Computers are commonly used for survey data analysis to compile and analyze the results of surveys and opinion polls. Figure 6.13 uses the array response initialized with 99 responses to a survey. Each response is a number from 1 to 9. The program computes the mean, median and mode of the 99 values. This example includes many common manipulations required in array problems, including passing arrays to functions. Notice that lines 48–52 contain several string literals separated only by whitespace. C compilers automatically combine such string literals into one—this helps making long string literals more readable.

```
// fig06_13.c
// Survey data analysis with arrays:
// computing the mean, median and mode of the data.
#include <stdio.h>
#define SIZE 99

// function prototypes
void mean(const int answer[]);
void median(int answer[]);
void mode(int freq[], const int answer[]);
void bubbleSort(int a[]);
void printArray(const int a[]);
```

Fig. 6.13 Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 1 of 5.)

```
13
14
   // function main begins program execution
15
   int main(void) {
       int frequency[10] = {0}; // initialize array frequency
16
17
18
       // initialize array response
19
       int response[SIZE] =
          {6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
20
21
           7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
22
           6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
           7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
23
24
           6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
           7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
25
26
           5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
27
           7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
28
           7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
29
           4, 5, 6, 1, 6, 5, 7, 8, 7};
30
31
       // process responses
32
       mean(response);
33
       median(response);
       mode(frequency, response);
34
35
   }
36
   // calculate average of all response values
37
   void mean(const int answer[]) {
38
       printf("%s\n%s\n", "-----", " Mean", "-----");
39
40
41
       int total = 0; // variable to hold sum of array elements
42
43
       // total response values
44
       for (size_t j = 0; j < SIZE; ++j) {
          total += answer[j];
45
46
       }
47
       printf("The mean is the average value of the data\n"
48
              "items. The mean is equal to the total of\n"
49
50
              "all the data items divided by the number\n"
51
              "of data items (%u). The mean value for\n"
              "this run is: %u / %u = %.4f\n\n",
52
53
              SIZE, total, SIZE, (double) total / SIZE);
54
   }
55
56
   // sort array and determine median element's value
57
    void median(int answer[]) {
       printf("\n%s\n%s\n%s\n%s", "-----", " Median", "-----",
58
59
              "The unsorted array of responses is");
60
61
       printArray(answer); // output unsorted array
62
63
       bubbleSort(answer); // sort array
```

Fig. 6.13 Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 2 of 5.)

```
64
       printf("%s", "\n\nThe sorted array is");
65
66
       printArray(answer); // output sorted array
67
       // display median element
68
       printf("\n\nThe median is element %u of\n"
69
               "the sorted %u element array.\n"
70
               "For this run the median is %u\n\n",
71
               SIZE / 2, SIZE, answer[SIZE / 2]);
72
73
   }
74
75
    // determine most frequent response
76
    void mode(int freq[], const int answer[]) {
       printf("\n%s\n%s\n", "-----", " Mode", "-----");
77
78
79
       // initialize frequencies to 0
80
       for (size_t rating = 1; rating <= 9; ++rating) {</pre>
81
          freq[rating] = 0;
82
       }
83
84
       // summarize frequencies
       for (size_t j = 0; j < SIZE; ++j) {
85
          ++freq[answer[j]];
86
87
       }
88
89
       // output headers for result columns
90
       printf("%s%11s%19s\n\n%54s\n%54s\n\n",
               "Response", "Frequency", "Bar Chart", "1 1 2 2", "5 0 5 0
91
92
                                                          5");
93
94
       // output results
95
       int largest = 0; // represents largest frequency
96
       int modeValue = 0; // represents most frequent response
97
98
       for (size_t rating = 1; rating <= 9; ++rating) {</pre>
          printf("%8zu%11d
                                      ", rating, freq[rating]);
99
100
           // keep track of mode value and largest frequency value
101
          if (freq[rating] > largest) {
102
             largest = freq[rating];
103
104
             modeValue = rating;
          }
105
106
107
          // output bar representing frequency value
108
          for (int h = 1; h <= freq[rating]; ++h) {</pre>
             printf("%s", "*");
109
          }
110
Ш
112
          puts(""); // being new line of output
113
       }
114
```

Fig. 6.13 Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 3 of 5.)

// display the mode value

115

```
printf("\nThe mode is the most frequent value.\n"
116
               "For this run the mode is %d which occurred %d times.\n",
117
118
              modeValue, largest);
119 }
120
121 // function that sorts an array with bubble sort algorithm
122 void bubbleSort(int a[]) {
       // loop to control number of passes
123
124
       for (int pass = 1; pass < SIZE; ++pass) {</pre>
          // loop to control number of comparisons per pass
125
126
          for (size_t i = 0; i < SIZE - 1; ++i) {
127
             // swap elements if out of order
128
              if (a[j] > a[j + 1]) {
                 int hold = a[j];
129
                 a[j] = a[j + 1];
130
131
                 a[j + 1] = hold;
132
             }
133
          }
134
       }
135 }
136
137 // output array contents (20 values per row)
138 void printArray(const int a[]) {
139
       // output array contents
140
       for (size_t j = 0; j < SIZE; ++j) {
141
          if (j \% 20 == 0) \{ // \text{ begin new line every } 20 \text{ values} \}
142
143
             puts("");
144
          }
145
146
          printf("%2d", a[j]);
147
       }
148 }
  Mean
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788
 Median
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
```

Fig. 6.13 Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 4 of 5.)

```
6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7
The sorted array is
1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5 5
5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
The median is element 49 of
the sorted 99 element array.
For this run the median is 7
 Mode
_____
Response Frequency
                      Bar Chart
                              1 1 2 2
0 5 0 5
     1
     2
              3
     3
                       ****
              5
                       ****
     4
     5
              8
     6
              9
                       ******
     7
              23
                       ********
     8
              27
                       **********
              19
                       *******
The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

Fig. 6.13 Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 5 of 5.)

Mean

The mean is the arithmetic average of the 99 values. Function mean (lines 38–54) computes the mean by totaling the 99 elements and dividing the result by 99.

Median

The median is the middle value. Function median (lines 57–73) first sorts the responses by calling function bubbleSort (defined in lines 122–135). Then it determines the median by picking the sorted array's middle element, answer[SIZE / 2]. When the number of elements is even, the median should be calculated as the mean of the two middle elements—function median does not currently provide this capability. Lines 61 and 66 call function printArray (lines 138–148) to output the response array before and after the sort.

Mode

The mode is the value that occurs most frequently among the 99 responses. Function mode (lines 76–119) determines the mode by counting the number of responses of each type, then selecting the value with the greatest count. This version of function mode does not handle a tie (see Exercise 6.14). Function mode also produces a bar chart to aid in determining the mode graphically.

Related Exercises

This case study is supported by the following exercise:

• Exercise 6.14 (Mean, Median and Mode Program Modifications).

Self Check

- [(Multiple Choice) Which of the following statements a), b) or c) is false?
 - a) The median is the middle value among the sorted data items.
 - b) The algorithm for finding the median of the values in an array sorts the array into ascending order, then picks the sorted array's middle element.
 - c) When the number of elements is even, the median is calculated as the mean of the two middle elements.
 - d) All of the above statements are *true*.

Answer: d.

- **2** (*Multiple Choice*) Which of the following statements is a), b) or c) is *false*?
 - a) The mode is the value that occurs most frequently among the data items.
 - b) The algorithm for finding the mode counts the number of occurrences of each value, then selects the most frequently occurring value. One problem with determining the mode is what to do in case of a tie.
 - c) The mode can be determined visually by graphing the frequencies of the values in a bar chart—the longest bar represents the mode.
 - d) All of the above statements are true.

Answer: d.

6.10 Searching Arrays

You'll often work with large amounts of data stored in arrays. It may be necessary to determine whether an array contains a value that matches a certain key value. The process of finding a key value in an array is called searching. This section discusses two searching techniques—the simple linear search technique and the more efficient (but more complicated) binary search technique. Exercises 6.32 and 6.33 ask you to implement recursive versions of the linear search and the binary search.

6.10.1 Searching an Array with Linear Search

A linear search (Fig. 6.14, lines 37–46) compares each array element with the search key. The array is not sorted, so it's just as likely the value will be found in the first element as in the last. On average, therefore, the program will have to compare the

search key with *half* the array elements. If the key value is found, we return the element's subscript; otherwise, we return -1 (an invalid subscript).

```
// fig06_14.c
 2
    // Linear search of an array.
 3 #include <stdio.h>
 4 #define SIZE 100
 5
 6
    // function prototype
 7
    int linearSearch(const int array[], int key, size_t size);
 8
 9
    // function main begins program execution
10
    int main(void) {
       int a[SIZE] = {0}; // create array a
11
12
13
       // create some data
14
       for (size_t x = 0; x < SIZE; ++x) {
15
          a[x] = 2 * x;
       }
16
17
18
       printf("Enter integer search key: ");
       int searchKey = 0; // value to locate in array a
19
20
       scanf("%d", &searchKey);
21
22
       // attempt to locate searchKey in array a
23
       int subscript = linearSearch(a, searchKey, SIZE);
24
25
       // display results
26
       if (subscript !=-1) {
          printf("Found value at subscript %d\n", subscript);
27
       }
28
29
       else {
30
          puts("Value not found");
31
32
    }
33
    // compare key to every element of array until the location is found
   // or until the end of array is reached; return subscript of element
35
    // if key is found or -1 if key is not found
36
37
    int linearSearch(const int array[], int key, size_t size) {
38
       // loop through array
39
       for (size_t n = 0; n < size; ++n) {
40
          if (array[n] == key) {
41
             return n; // return location of key
42
       }
43
44
45
       return -1; // key not found
46
    }
```

```
Enter integer search key: 36
Found value at subscript 18
```

Fig. 6.14 | Linear search of an array. (Part 1 of 2.)

Enter integer search key: 37 Value not found

Fig. 6.14 | Linear search of an array. (Part 2 of 2.)

6.10.2 Searching an Array with Binary Search

Linear searching works well for small or unsorted arrays. However, for large arrays, linear searching is inefficient. If the array is sorted, the high-speed binary search technique can be used.

The binary search algorithm eliminates from consideration *one-half* of a sorted array's elements after each comparison. The algorithm locates the middle array element and compares it to the search key. If they're equal, the algorithm found the search key, so it returns that element's subscript. If they're not equal, the problem is reduced to searching *one-half* of the array. If the search key is less than the middle array element, the algorithm searches the *first half* of the array; otherwise, it searches the *second half*. If the search key is not the middle element in the current subarray (a piece of the original array), the algorithm repeats on one-quarter of the original array. The search continues until the search key is equal to the middle element of a subarray, or until the subarray consists of one element that's not equal to the search key—that is, the search key is not found.

PERF * Performance of the Binary Search Algorithm

In the worst-case scenario, searching a sorted array of 1023 elements takes only 10 comparisons using a binary search. Repeatedly dividing 1,024 by 2 yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1. The number 1,024 (210) is divided by 2 only 10 times to get the value 1. Dividing by 2 is equivalent to one comparison in the binary search algorithm. An array of 1,048,576 (220) elements takes a maximum of only 20 comparisons to find the search key. A sorted array of one billion elements takes a maximum of only 30 comparisons to find the search key. This is a tremendous increase in performance over a linear search of a sorted array, which requires comparing the search key to an average of half of the array elements. For a one-billion-element array, this is a difference between an average of 500 million comparisons and a maximum of 30 comparisons! The maximum comparisons for any array can be determined by finding the first power of 2 greater than the number of array elements.

Implementing Binary Search

Figure 6.15 presents the iterative version of function binarySearch (lines 39–60). The function receives four arguments—an integer array b to search, an integer key to find, the 1ow array subscript and the high array subscript. The last two arguments define the portion of the array to search. If the search key does not match the middle element of a subarray, the 1ow subscript or high subscript is modified so that a smaller subarray can be searched:

- If the search key is less than the middle element, the algorithm sets the high subscript to middle 1 (line 52), then continues the search on the elements with subscripts in the range low to middle 1.
- If the search key is greater than the middle element, the algorithm sets the low subscript to middle + 1 (line 55), then continues the search on the elements with subscripts in the range middle + 1 to high.

The program uses an array of 15 elements. The first power of 2 greater than the number of elements in this array is 16 (24), so no more than 4 comparisons are required to find the search key. We use function printHeader (lines 63–79) to output the array subscripts and function printRow (lines 83–99) to output each subarray during the binary search process. We mark the middle element in each subarray with an asterisk (*) to indicate the element to which the search key is compared.

```
I // fig06_15.c
 2 // Binary search of a sorted array.
 3 #include <stdio.h>
 4 #define SIZE 15
 5
 6
   // function prototypes
 7
   int binarySearch(const int b[], int key, size_t low, size_t high);
    void printHeader(void);
   void printRow(const int b[], size_t low, size_t mid, size_t high);
 9
10
    // function main begins program execution
11
12
    int main(void) {
       int a[SIZE] = {0}; // create array a
13
14
15
       // create data
16
       for (size_t i = 0; i < SIZE; ++i) {
          a[i] = 2 * i;
17
18
19
       printf("%s", "Enter a number between 0 and 28: ");
20
21
       int key = 0; // value to locate in array a
22
       scanf("%d", &key);
23
24
       printHeader();
25
26
       // search for key in array a
27
       int result = binarySearch(a, key, 0, SIZE - 1);
28
29
       // display results
       if (result !=-1) {
30
31
          printf("\n%d found at subscript %d\n", key, result);
32
       }
33
       else {
          printf("\n%d not found\n", key);
34
35
       }
36
    }
```

Fig. 6.15 Binary search of a sorted array. (Part 1 of 3.)

```
37
38
    // function to perform binary search of an array
    int binarySearch(const int b[], int key, size_t low, size_t high) {
39
       // loop until low subscript is greater than high subscript
40
       while (low <= high) {</pre>
41
          size_t middle = (low + high) / 2; // determine middle subscript
42
43
44
          // display subarray used in this loop iteration
          printRow(b, low, middle, high);
45
46
47
          // if key matches, return middle subscript
48
          if (key == b[middle]) {
49
             return middle;
50
          else if (key < b[middle]) { // if key < b[middle], adjust high</pre>
51
52
             high = middle - 1; // next iteration searches low end of array
53
54
          else { // key > b[middle], so adjust low
55
             low = middle + 1; // next iteration searches high end of array
56
57
       } // end while
58
59
       return -1; // searchKey not found
   }
60
61
62
   // Print a header for the output
63
   void printHeader(void) {
64
       puts("\nSubscripts:");
65
66
       // output column head
67
       for (int i = 0; i < SIZE; ++i) {
68
          printf("%3d ", i);
69
       }
70
71
       puts(""); // start new line of output
72
73
       // output line of - characters
       for (int i = 1; i <= 4 * SIZE; ++i) {
74
          printf("%s", "-");
75
76
77
78
       puts(""); // start new line of output
79
    }
80
81
    // Print one row of output showing the current
82
    // part of the array being processed.
83
    void printRow(const int b[], size_t low, size_t mid, size_t high) {
84
       // loop through entire array
85
       for (size_t i = 0; i < SIZE; ++i) {
          // display spaces if outside current subarray range
86
          if (i < low || i > high) {
87
             printf("%s", "
88
                                ");
89
          }
```

Fig. 6.15 | Binary search of a sorted array. (Part 2 of 3.)

```
90
          else if (i == mid) { // display middle element
             printf("%3d*", b[i]); // mark middle value
91
92
          else { // display other elements in subarray
93
94
             printf("%3d ", b[i]);
95
          }
       }
96
97
98
       puts(""); // start new line of output
    }
99
```

```
Enter a number between 0 and 28: 25

Subscripts:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

0 2 4 6 8 10 12 14* 16 18 20 22 24 26 28

16 18 20 22* 24 26 28

24 26* 28

25 not found
```

```
Enter a number between 0 and 28: 8

Subscripts:

0  1  2  3  4  5  6  7  8  9  10  11  12  13  14

0  2  4  6  8  10  12  14* 16  18  20  22  24  26  28

0  2  4  6*  8  10  12  8  10* 12  8  8*

8 found at subscript 4
```

```
Enter a number between 0 and 28: 6

Subscripts:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

0 2 4 6 8 10 12 14* 16 18 20 22 24 26 28
0 2 4 6* 8 10 12

6 found at subscript 3
```

Fig. 6.15 | Binary search of a sorted array. (Part 3 of 3.)

Self Check

- I (Multiple Choice) Which of the following statements about linear search is false?
 - a) It compares every array element with the search key.
 - b) Because the array is not in any particular order, it's just as likely that the value will be found in the first element as in the last.
 - c) On average, it compares the search key with half the array's elements.

- d) It works well for small or unsorted arrays. For *large* arrays it's *inefficient*. **Answer:** a) is *false*. It could find a match before reaching the array's end, in which case it would terminate the search before comparing every element with the search key.
- **2** (*Multiple Choice*) Which of the following statements about binary search is *false*?
 - a) If the array is sorted, the high-speed binary search technique can be used.
 - b) The binary search algorithm eliminates from consideration two of the elements in a sorted array after each comparison.
 - c) The algorithm locates the middle element of the array and compares it to the search key. If they're equal, the search key is found and the array index of that element is returned. If the search key is less than the middle element of the array, the algorithm then searches the array's first half; otherwise, the algorithm searches the array's second half.
 - d) The search continues until the search key is equal to a subarray's middle element, or until the subarray consists of one element that's not equal to the search key (i.e., the search key is not found).

Answer: b) is *false*. Actually, the binary search algorithm eliminates from consideration one-half of the elements in a sorted array after each comparison.

6.11 Multidimensional Arrays

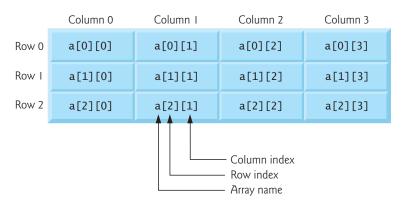
Arrays can have multiple subscripts. A common use of multidimensional arrays is to represent tables of values consisting of information arranged in *rows* and *columns*. To identify a particular table element, we specify two subscripts:

- The *first* (by convention) identifies the element's *row* and
- the *second* (by convention) identifies the element's *column*.

Arrays that require two subscripts to identify a particular element commonly are called two-dimensional arrays. Multidimensional arrays can have more than two subscripts.

6.11.1 Illustrating a Two-Dimensional Array

The following diagram illustrates a two-dimensional array named a:



The array contains three rows and four columns, so it's said to be a 3-by-4 array. In general, an array with m rows and n columns is called an m-by-n array.

Every element in array a is identified by a name of the form a[i][j], where a is the array name, and i and j are the subscripts that uniquely identify each element. The element names in row 0 all have the first subscript 0. The element names in column 3 all have the second subscript 3. Referencing a two-dimensional array element (X) FRR as a[x, y] instead of a[x][y] is a logic error. C treats a[x, y] as a[y], so this programmer error is not a *syntax* error. The comma in this context is a comma operator which guarantees that a list of expressions evaluates from left to right. The value of a commaseparated list of expressions is the value of the rightmost expression in the list.



6.11.2 Initializing a Double-Subscripted Array

You can initialize a multidimensional array when you define it. For example, you can define and initialize the two-dimensional array int b[2][2] with:

```
int b[2][2] = \{\{1, 2\}, \{3, 4\}\};
```

The values in the initializer list are grouped by row in braces. The values in the first set of braces initialize row 0 and the values in the second set of braces initialize row 1. So, the values 1 and 2 initialize elements b[0][0] and b[0][1], respectively, and the values 3 and 4 initialize elements b[1][0] and b[1][1], respectively. If there are not enough initializers for a given row, that row's remaining elements are initialized to 0. So the definition:

```
int b[2][2] = \{\{1\}, \{3, 4\}\};
```

would initialize b[0][0] to 1, b[0][1] to 0, b[1][0] to 3 and b[1][1] to 4. Figure 6.16 demonstrates defining and initializing two-dimensional arrays.

```
// fig06_16.c
    // Initializing multidimensional arrays.
 2
 3
   #include <stdio.h>
 5
    void printArray(int a[][3]); // function prototype
 6
 7
    // function main begins program execution
 8
    int main(void) {
       int array1[2][3] = \{\{1, 2, 3\}, \{4, 5, 6\}\};
 9
       puts("Values in array1 by row are:");
10
       printArray(array1);
11
12
       int array2[2][3] = \{\{1, 2, 3\}, \{4, 5\}\};
13
       puts("Values in array2 by row are:");
14
15
       printArray(array2);
16
17
       int array3[2][3] = \{\{1, 2\}, \{4\}\};
       puts("Values in array3 by row are:");
18
       printArray(array3);
19
20
    }
```

Initializing multidimensional arrays. (Part 1 of 2.)

```
21
    // function to output array with two rows and three columns
22
void printArray(int a[][3]) {
       // loop through rows
24
       for (size_t i = 0; i <= 1; ++i) {
25
          // output column values
26
          for (size_t j = 0; j \le 2; ++j) {
27
             printf("%d ", a[i][j]);
28
          }
29
30
          printf("\n"); // start new line of output
31
       }
32
33
    }
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Fig. 6.16 | Initializing multidimensional arrays. (Part 2 of 2.)

array1 Definition

The program defines three arrays of two rows and three columns. The definition of array1 (line 9) provides six initializers in two sublists. The first sublist initializes *row* 0 to the values 1, 2 and 3, and the second sublist initializes *row* 1 to the values 4, 5 and 6.

array2 Definition

The definition of array2 (line 13) provides five initializers in two sublists, initializing *row 0* to 1, 2 and 3, and *row 1* to 4, 5 and 0. Any elements that do not have an explicit initializer are initialized to zero automatically, so array2[1][2] is initialized to 0.

array3 Definition

The definition of array3 (line 17) provides three initializers in two sublists. The first row's sublist explicitly initializes the row's first two elements to 1 and 2 and implicitly initializes the third element to 0. The second row's sublist explicitly initializes the first element to 4 and implicitly initializes the last two elements to 0.

printArray Function

The program calls printArray (lines 23–33) to output each array's elements. The function definition specifies the array parameter as int a[][3]. In a one-dimensional array parameter, the array brackets are empty. The first subscript of a multidimensional array is not required, but all subsequent subscripts are required. The compiler

uses these subscripts to determine the locations in memory of a multidimensional array's elements. All array elements are stored consecutively in memory regardless of the number of subscripts. In a two-dimensional array, the first row is stored in memory, followed by the second row.

Providing the subscript values in a parameter declaration enables the compiler to tell the function how to locate an array element. In a two-dimensional array, each row is basically a one-dimensional array. To locate an element in a particular row, the compiler must know how many elements are in each row so that it can skip the proper number of memory locations when accessing the array. So, when accessing a[1][2] in our example, the compiler knows to skip the three elements of the first row to get to the second row (row 1). Then, the compiler accesses element 2 of that row.

6.11.3 Setting the Elements in One Row

Many common array manipulations use for iteration statements. For example, the following statement sets all the elements in row 2 of the 3-by-4 int array a to zero:

```
for (int column = 0; column <= 3; ++column) {
    a[2][column] = 0;
}</pre>
```

We specified row 2, so the first subscript is always 2. The 100p varies only the column subscript. The preceding for statement is equivalent to the assignment statements:

```
a[2][0] = 0;
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

6.11.4 Totaling the Elements in a Two-Dimensional Array

The following nested for statement totals the elements in the 3-by-4 int array a:

```
int total = 0;
for (int row = 0; row <= 2; ++row) {
   for (int column = 0; column <= 3; ++column) {
     total += a[row][column];
   }
}</pre>
```

The for statement totals the elements one row at a time. The outer for statement begins by setting the row subscript to 0 so that row's elements may be totaled by the inner for statement. The outer for statement then increments row to 1, so that row's elements can be totaled. Then, the outer for statement increments row to 2, so that row's elements can be totaled. When the nested for statement terminates, total contains the sum of all the elements in the array a.

6.11.5 Two-Dimensional Array Manipulations

Figure 6.17 uses for statements to perform several common array manipulations on a 3-by-4 array named studentGrades. Each row represents a student, and each col-

umn represents a grade on one of the four exams the students took during the semester. The array manipulations are performed by four functions:

- Function minimum (lines 38–52) finds the lowest grade of any student for the semester.
- Function maximum (lines 55–69) finds the highest grade of any student for the semester.
- Function average (lines 72–81) calculates a particular student's semester average.
- Function printArray (lines 84–98) displays the two-dimensional array in a neat, tabular format.

```
I // fig06_17.c
 2 // Two-dimensional array manipulations.
 3 #include <stdio.h>
 4 #define STUDENTS 3
 5 #define EXAMS 4
 6
 7 // function prototypes
   int minimum(const int grades[][EXAMS], size_t pupils, size_t tests);
 9 int maximum(const int grades[][EXAMS], size_t pupils, size_t tests);
    double average(const int setOfGrades[], size_t tests);
10
    void printArray(const int grades[][EXAMS], size_t pupils, size_t tests);
11
12
    // function main begins program execution
13
14
    int main(void) {
       // initialize student grades for three students (rows)
15
       int studentGrades[STUDENTS][EXAMS] =
16
17
          {{77, 68, 86, 73},
18
           {96, 87, 89, 78},
           {70, 90, 86, 81}};
19
20
21
       // output array studentGrades
22
       puts("The array is:");
       printArray(studentGrades, STUDENTS, EXAMS);
23
24
       // determine smallest and largest grade values
25
26
       printf("\n\nLowest grade: %d\nHighest grade: %d\n",
27
          minimum(studentGrades, STUDENTS, EXAMS),
          maximum(studentGrades, STUDENTS, EXAMS));
28
29
30
       // calculate average grade for each student
       for (size_t student = 0; student < STUDENTS; ++student) {</pre>
31
          printf("The average grade for student %zu is %.2f\n",
32
             student, average(studentGrades[student], EXAMS));
33
34
35
    }
```

Fig. 6.17 Two-dimensional array manipulations. (Part 1 of 3.)

```
36
    // Find the minimum grade
37
    int minimum(const int grades[][EXAMS], size_t pupils, size_t tests) {
38
       int lowGrade = 100; // initialize to highest possible grade
39
40
       // loop through rows of grades
41
       for (size_t row = 0; row < pupils; ++row) {</pre>
42
          // loop through columns of grades
43
44
          for (size_t column = 0; column < tests; ++column) {</pre>
             if (grades[row][column] < lowGrade) {</pre>
45
                 lowGrade = grades[row][column];
46
47
             }
48
          }
49
       }
50
51
       return lowGrade; // return minimum grade
52
    }
53
54
    // Find the maximum grade
    int maximum(const int grades[][EXAMS], size_t pupils, size_t tests) {
55
56
       int highGrade = 0; // initialize to lowest possible grade
57
       // loop through rows of grades
58
59
       for (size_t row = 0; row < pupils; ++row) {</pre>
60
          // loop through columns of grades
          for (size_t column = 0; column < tests; ++column) {</pre>
61
62
             if (grades[row][column] > highGrade) {
63
                 highGrade = grades[row][column];
64
             }
65
          }
       }
66
67
68
       return highGrade; // return maximum grade
    }
69
70
    // Determine the average grade for a particular student
71
72
    double average(const int setOfGrades[], size_t tests) {
73
       int total = 0; // sum of test grades
74
75
       // total all grades for one student
76
       for (size_t test = 0; test < tests; ++test) {</pre>
77
          total += setOfGrades[test];
78
79
80
       return (double) total / tests; // average
81
    }
82
    // Print the array
83
84
    void printArray(const int grades[][EXAMS], size_t pupils, size_t tests) {
85
       // output column heads
       printf("%s", "
                                        [0] [1] [2] [3]");
86
87
```

Fig. 6.17 | Two-dimensional array manipulations. (Part 2 of 3.)

```
// output grades in tabular format
88
       for (size_t row = 0; row < pupils; ++row) {</pre>
89
          // output label for row
90
          printf("\nstudentGrades[%zu] ", row);
91
92
93
          // output grades for one student
          for (size_t column = 0; column < tests; ++column) {</pre>
94
95
             printf("%-5d", grades[row][column]);
96
       }
97
98
    }
The array is:
                  [0]
                       [1]
                             [2]
                                  [3]
studentGrades[0] 77
                                  73
                       68
                             86
                       87
                             89
                                  78
studentGrades[1] 96
studentGrades[2] 70
                       90
                             86
                                  81
Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
```

Fig. 6.17 Two-dimensional array manipulations. (Part 3 of 3.)

The average grade for student 2 is 81.75

Nested Loops in Functions minimum, maximum and printArray

Functions minimum, maximum and printArray each receive three arguments—the studentGrades array (called grades in each function), the number of students (rows in the array) and the number of exams (columns in the array). Each function loops through array grades using nested for statements. The following nested for statement is from the function minimum definition:

```
// loop through rows of grades
for (size_t row = 0; row < pupils; ++row) {
    // loop through columns of grades
    for (size_t column = 0; column < tests; ++column) {
        if (grades[row][column] < lowGrade) {
            lowGrade = grades[row][column];
        }
    }
}</pre>
```

The outer for statement begins by setting row to 0 so that row's elements (i.e., the first student's grades) can be compared to variable lowGrade in the inner for statement. The inner for statement loops through a particular row's four grades and compares each grade to lowGrade. If a grade is less than lowGrade, the nested if statement sets lowGrade to that grade. The outer for statement then increments row to 1, and that row's elements are compared to lowGrade. When the nested

statement completes execution, 10wGrade contains the smallest grade in the twodimensional array. Function maximum works similarly to function minimum.

Function average

Function average (lines 72–81) takes two arguments—a one-dimensional array of test results for a particular student (setOfGrades) and the number of test results in the array. When line 33 calls average, the first argument—studentGrades[student]—passes the address of one row of the two-dimensional array. The argument student-Grades[1] is the starting address of row 1 of the array. Remember that a two-dimensional array is basically an array of one-dimensional arrays, and the name of a one-dimensional array is the address of that array in memory. Function average calculates the sum of the array elements, divides the total by the number of test results and returns the floating-point result.

Self Check

I (What Does This Code Do?) What does the following nested for statement do?

```
product = 1;

for (row = 0; row <= 2; ++row) {
    for (column = 0; column <= 3; ++column) {
        product *= m[row][column];
     }
}</pre>
```

Answer: It calculates the product of all the element values in 3-by-4 double array m.

2 (What Does This Code Do?) What does the following nested for statement do?

```
// loop through rows of grades
for (i = 0; i < pupils; ++i) {
    // loop through columns of grades
    for (j = 0; j < tests; ++j) {
        if (grades[i][j] < lowGrade) {
            lowGrade = grades[i][j];
        }
    }
}</pre>
```

Answer: It loops through a two-dimensional array grades with pupils rows and tests columns attempting to find the minimum grade in the array. Assuming the grades would be zero through 100, lowGrade would need to be initialized to a value of 100 or greater.

6.12 Variable-Length Arrays

For each array you've defined so far, you've specified its size at compilation time. But what if you cannot determine an array's size until execution time? In the past, to handle this, you had to use dynamic memory allocation (introduced in Chapter 12, Data Structures). For cases in which an array's size is not known at compilation time, C

has variable-length arrays (VLAs)—arrays whose lengths are determined by expressions evaluated at execution time.² The program of Fig. 6.18 declares and prints several VLAs.

```
| // fig06_18.c
 2 // Using variable-length arrays in C99
 3 #include <stdio.h>
 5
    // function prototypes
  void print1DArray(size_t size, int array[size]);
 7 void print2DArray(size_t row, size_t col, int array[row][col]);
 8
 9
    int main(void) {
       printf("%s", "Enter size of a one-dimensional array: ");
10
П
       int arraySize = 0; // size of 1-D array
12
       scanf("%d", &arraySize);
13
14
       int array[arraySize]; // declare 1-D variable-length array
15
16
       printf("%s", "Enter number of rows and columns in a 2-D array: ");
17
       int row1 = 0; // number of rows in a 2-D array
18
       int col1 = 0; // number of columns in a 2-D array
19
       scanf("%d %d", &row1, &col1);
20
       int array2D1[row1][col1]; // declare 2-D variable-length array
21
22
23
       printf("%s",
24
          "Enter number of rows and columns in another 2-D array: ");
       int row2 = 0; // number of rows in a 2-D array
25
26
       int col2 = 0; // number of columns in a 2-D array
27
       scanf("%d %d", &row2, &col2);
28
29
       int array2D2[row2][col2]; // declare 2-D variable-length array
30
31
       // test sizeof operator on VLA
32
       printf("\nsizeof(array) yields array size of %zu bytes\n",
33
          sizeof(array));
34
35
       // assign elements of 1-D VLA
36
       for (size_t i = 0; i < arraySize; ++i) {</pre>
          array[i] = i * i;
37
38
       }
39
40
       // assign elements of first 2-D VLA
41
       for (size_t i = 0; i < row1; ++i) {
42
          for (size_t j = 0; j < col1; ++j) {
43
             array2D1[i][j] = i + j;
44
          }
45
       }
46
```

Fig. 6.18 Using variable-length arrays in C99. (Part 1 of 3.)

^{2.} This feature is not supported in Microsoft Visual C++.

```
// assign elements of second 2-D VLA
       for (size_t i = 0; i < row2; ++i) {
48
49
          for (size_t j = 0; j < col2; ++j) {
50
             array2D2[i][j] = i + j;
51
          }
       }
52
53
54
       puts("\n0ne-dimensional array:");
55
       print1DArray(arraySize, array); // pass 1-D VLA to function
56
       puts("\nFirst two-dimensional array:");
57
58
       print2DArray(row1, col1, array2D1); // pass 2-D VLA to function
59
60
       puts("\nSecond two-dimensional array:");
61
       print2DArray(row2, col2, array2D2); // pass other 2-D VLA to function
62
    }
63
64
    void print1DArray(size_t size, int array[size]) {
65
       // output contents of array
66
       for (size_t i = 0; i < size; i++) {
67
          printf("array[%zu] = %d\n", i, array[i]);
       }
68
69
    }
70
    void print2DArray(size_t row, size_t col, int array[row][col]) {
71
72
       // output contents of array
73
       for (size_t i = 0; i < row; ++i) {
74
          for (size_t j = 0; j < col; ++j) {
75
             printf("%5d", array[i][j]);
76
          }
77
78
          puts("");
79
       }
   }
80
Enter size of a one-dimensional array: 6
Enter number of rows and columns in a 2-D array: 2 5
Enter number of rows and columns in another 2-D array: 4 3
sizeof(array) yields array size of 24 bytes
One-dimensional array:
array[0] = 0
array[1] = 1
array[2] = 4
array[3] = 9
array[4] = 16
array[5] = 25
First two-dimensional array:
         1
              2
                   3
    1
         2
              3
                   4
                         5
```

Fig. 6.18 Using variable-length arrays in C99. (Part 2 of 3.)

47

Fig. 6.18 Using variable-length arrays in C99. (Part 3 of 3.)

Creating the VLAs

Lines 10–29 prompt the user for the desired sizes for a one-dimensional array and two two-dimensional arrays and use the input values in lines 14, 21 and 29 to create VLAs. These lines are valid as long as the variables representing the array sizes are integers.

sizeof Operator with VLAs

After creating the arrays, we use the sizeof operator in lines 32–33 to check our onedimensional VLA's length. Operator sizeof is normally a compile-time operation, but it operates at runtime when applied to a VLA. The output window shows that the sizeof operator returns a size of 24 bytes—four times the number we entered because the size of an int on our machine is 4 bytes.

Assigning Values to VLA Elements

Next, we assign values to our VLAs' elements (lines 36–52). We use the loop-continuation condition i < arraySize when filling the one-dimensional array. As with fixed-length arrays, there's no protection against stepping outside the array bounds.

Function print1DArray

Lines 64–69 define function print1DArray that displays its one-dimensional VLA argument. VLA function parameters have the same syntax as regular array parameters. We use the parameter size in parameter array's declaration, but it's purely documentation for the programmer.

Function print2DArray

Function print2DArray (lines 71–80) displays a two-dimensional VLA. Recall that you must specify a size for all but the first subscript in a multidimensional array parameter. The same restriction holds true for VLAs, except that the sizes can be specified by variables. The initial value of col passed to the function determines where each row begins in memory, just as with a fixed-size array.

✓ Self Check

I (*TruelFalse*) Unlike with fixed-length arrays, VLAs offer protection against stepping outside the array bounds.

Answer: *False.* Actually, as with fixed-length arrays, there is no protection against stepping outside the array bounds.

2 (*True/False*) sizeof is a compile-time-only operation.

Answer: *False.* Actually, generally sizeof is a compile-time operation, but when applied to a VLA, sizeof operates at runtime.

6.13 Secure C Programming



Bounds Checking for Array Subscripts

It's important to ensure that every subscript used to access an array element is within the array's bounds. A one-dimensional array's subscripts must be greater than or equal to 0 and less than the number of elements. A two-dimensional array's row and column subscripts must be greater than or equal to 0 and less than the numbers of rows and columns, respectively. This also applies to arrays with additional dimensions.

Allowing programs to read from or write to array elements outside an array's bounds are common security flaws. Reading from out-of-bounds array elements can cause a program to crash or even appear to execute correctly while using bad data. Writing to an out-of-bounds element (known as a buffer overflow) can corrupt a program's data in memory, crash a program and even allow attackers to exploit the system and execute their own code.

C provides no automatic bounds checking for arrays. You must ensure that array subscripts are always greater than or equal to 0 and less than the array's number elements. For additional techniques that help you prevent such problems, see CERT guideline ARR30-C at https://wiki.sei.cmu.edu/confluence.

scanf_s

Bounds checking is also important in string processing. When reading a string into a char array, scanf does not automatically prevent buffer overflows. If the number of characters input is greater than or equal to the array's length, scanf will write characters—including the string's terminating null character ('\0')—beyond the end of the array. This might overwrite other variables' values in memory. In addition, if the program writes to those other variables, it might overwrite the string's '\0'.

A function determines where a string ends by looking for its terminating '\0' character. For example, recall that function printf outputs a string by reading characters from the beginning of the string in memory and continuing until it encounters the string's '\0'. If the '\0' is missing, printf continues reading from memory (and printing) until it encounters some later '\0' in memory. This can lead to strange results or cause a program to crash.

The C11 standard's optional Annex K provides more secure versions of many string-processing and input/output functions. When reading a string into a character array, function scanf_s performs checks to ensure that it does not write beyond the end of the array. Assuming that myString is a 20-character array, the statement

```
scanf_s("%19s", myString, 20);
```

reads a string into myString. Function scanf_s requires two arguments for each %s in the format string:

- a character array in which to place the input string and
- the array's number of elements.

The function uses the number of elements to prevent buffer overflows. For example, it's possible to supply a field width for %s that's too long for the underlying character array, or to simply omit the field width entirely. With scanf_s, if the number of characters input plus the terminating null character is larger than the number of array elements specified, the %s conversion fails. For the preceding statement, which contains only one conversion specification, scanf_s would return 0, indicating no conversions were performed. The array myString would be unaltered. We discuss additional Annex K functions in later Secure C Programming sections.

Not all compilers support the C11 standard's Annex K functions. For programs that must compile on multiple platforms and compilers, you might have to edit your code to use the versions of scanf_s or scanf available on each platform. Your compiler might also require a specific setting to enable you to use the Annex K functions.

Don't Use Strings Read from the User as Format-Control Strings

You might have noticed that throughout this book, we do not use single-argument printf statements. Instead, we use one of the following forms:

• When we need to output a '\n' after the string, we use function puts (which automatically outputs a '\n' after its single string argument), as in

```
puts("Welcome to C!");
```

• When we need the cursor to remain on the same line as the string, we use function printf, as in

```
printf("%s", "Enter first integer: ");
```

Because we were displaying string literals, we certainly could have used the one-argument form of printf, as in

```
printf("Welcome to C!\n");
printf("Enter first integer: ");
```

When printf evaluates the format-control string in its first (and possibly only) argument, it performs tasks based on the conversion specification(s) in that string. If the format-control string were obtained from the user, an attacker could supply malicious conversion specifications that would be "executed" by the formatted output function. Now that you know how to read strings into character arrays, it's important to note that you should never use as a printf's format-control string a character array that might contain user input. For more information, see CERT guideline FIO30-C at https://wiki.sei.cmu.edu/confluence.

Self Check

- [(Multiple Choice) Which of the following statements a), b) or c) is false?
 - a) It's important to ensure that every index you use to access an array element is within the array's bounds. A one-dimensional array's indexes must be greater

- than or equal to 0 and less than the number of array elements. A two-dimensional array's row and column indexes must be greater than or equal to 0 and less than the numbers of rows and columns, respectively.
- b) C provides no automatic bounds checking for arrays, so you must provide your own. Allowing programs to read from or write to array elements outside the bounds of arrays are common security flaws.
- c) Reading from out-of-bounds array elements can cause a program to crash or even appear to execute correctly while using bad data.
- d) All of the above statements are *true*.

Answer: d.

2 (*True/False*) When printf evaluates the format-control string in its first (and possibly its only) argument, the function performs tasks based on the conversion specifier(s) in that string. If the format-control string were obtained from the user, an attacker could supply malicious conversion specifiers that would be "executed" by the formatted output function. You should never use as a printf's format-control string a character array that might contain user input.

Answer: True.

Summary

Section 6.1 Introduction

- Arrays (p. 244) are data structures consisting of related data items of the same type.
- Arrays are "static" entities in that they remain the same size throughout program execution.

Section 6.2 Arrays

- An array is a contiguous group of memory locations related by the fact that they all have the same name and the same type.
- To refer to a particular **location** or **element** (p. 244), specify the array's name and the **position number** (p. 245) of the element.
- The first element in every array is at location 0.
- The position number contained within square brackets is more formally called a **subscript** (p. 245) or **index**. A subscript must be an integer or an integer expression.
- The **brackets** that enclose an array subscript are an operator with the highest level of precedence.

Section 6.3 Defining Arrays

- You specify the array's element type and **number of elements** so that the computer may reserve the appropriate amount of memory.
- An array of type char can be used to store a character string.

Section 6.4 Array Examples

• Type size_t represents an unsigned integral type. This type is recommended for any variable that represents an array's size or an array's subscripts. The header <stddef.h> defines size_t and is often included by other headers (such as <stdio.h>).

- An array's elements can be initialized when the array is defined by following the definition with an equals sign and braces, {}, containing a comma-separated list of initializers (p. 248). If there are fewer initializers than array elements, the remaining elements are initialized to zero.
- The statement int n[10] = {0}; explicitly initializes the first element to zero and initializes the remaining nine elements to zero because there are fewer initializers than array elements.
- If the array size is omitted from a definition with an initializer list, the compiler determines the array's number of elements from the number of initializers.
- The #define preprocessor directive can define a symbolic constant—an identifier that the preprocessor replaces with replacement text before the program is compiled. When a program is preprocessed, all occurrences of the symbolic constant are replaced with the replacement text (p. 249). Using symbolic constants to specify array sizes makes programs easier to read and more modifiable.
- C has **no** array bounds checking to prevent a program from referring to an element that does not exist. Thus, an executing program can "walk off" the end of an array without warning. You should ensure that all array references remain within the bounds of the array.

Section 6.5 Using Character Arrays to Store and Manipulate Strings

- A string literal such as "hello" is really an array of individual characters in C.
- A character array can be initialized using a string literal. In this case, the array's size is determined by the compiler based on the string's length.
- Every string contains a special string-termination character called the null character (p. 255). The character constant representing the null character is '\0'.
- A character array representing a string should always be defined large enough to hold the number of characters in the string and the terminating null character.
- Character arrays also can be initialized with individual character constants in an initializer list.
- Because a string is really an array of characters, we can access individual characters in a string directly using array subscript notation.
- You can input a string directly into a character array from the keyboard using scanf and the **conversion specification %s.** The character array's name is passed to scanf without the preceding & used with non-array variables.
- Function scanf reads characters from the keyboard until the first whitespace character is encountered—it does not check the array size. Thus, scanf can write beyond the end of an array. For this reason, when reading a string into a char array with scanf, you should always use a field width that's one less than the char array's size (e.g., "%19s" for a 20-char array).
- A character array representing a string can be output with printf and the %s conversion specification. The characters of the string are printed until a terminating null character is encountered.

Section 6.6 Static Local Arrays and Automatic Local Arrays

- We can apply static to a local array definition so the function does not create and initialize
 the array in each call and destroy it each time the function exits. This reduces program execution time, particularly for programs with frequently called functions that contain large
 arrays.
- Arrays that are static are automatically initialized once at program startup. If you do not
 explicitly initialize a static array, that array's elements are initialized to zero by the compiler.

Section 6.7 Passing Arrays to Functions

- To pass an array argument to a function, specify the array name without any brackets.
- Unlike char arrays that contain strings, other array types do not have a special terminator. For this reason, the array's size is passed to a function, so the function can process the proper number of elements.
- C automatically passes arrays to functions by reference—the called functions can modify the element values in the callers' original arrays. An array name evaluates to the address of the array's first element. Because the starting address of the array is passed, the called function knows precisely where the array is stored and can modify the original array in the caller.
- Although entire arrays are passed by reference, individual array elements are passed by value exactly as simple variables are.
- Single pieces of data (such as individual ints, floats and chars) are called scalars (p. 261).
- To pass an array element to a function, use the subscripted name of the array element.
- For a function to receive an array through a function call, the function's parameter list must specify that an array will be received. The array's size is not required between the array brackets. If it's included, the compiler checks that it's greater than zero, then ignores it.
- If an array parameter is preceded by the **const qualifier** (p. 263), any attempt to modify an element in the function body results in a compilation error.

Section 6.8 Sorting Arrays

- Sorting data—that is, placing the data into ascending or descending order—is one of the most important computing applications.
- One sorting technique is called the **bubble sort** (p. 265) or the **sinking sort**, because the smaller values gradually "bubble" their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom of the array. The technique is to make several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array.
- Bubble sort may move a large value down the array many positions on a single pass but may move a small value up only one position.
- The chief virtue of the bubble sort is that it's easy to program. However, it runs slowly. This becomes apparent when sorting large arrays.

Section 6.9 Intro to Data Science Case Study: Survey Data Analysis

- The **mean** is the arithmetic average of a set of values.
- The **median** is the "middle value" in a sorted set of values.
- The **mode** is the value that occurs most frequently in a set of values.

Section 6.10 Searching Arrays

- The process of finding a particular array element is called **searching** (p. 272).
- The linear search compares each array element with a search key (p. 272). The array is not in any particular order, so it's just as likely that the value will be found in the first element as in the last. On average, therefore, the search key will be compared with half the array's elements.
- The linear search algorithm (p. 272) works well for small or unsorted arrays. For sorted arrays, the high-speed binary search algorithm can be used.

- The binary search algorithm (p. 272) eliminates from consideration one-half of a sorted array's elements after each comparison. The algorithm locates the middle array element and compares it to the search key. If they're equal, the search key is found, and that element's subscript is returned. If they're not equal, the problem is reduced to searching one-half of the array. If the search key is less than the middle array element, the array's first half is searched; otherwise, the second half is searched. If the search key is not found in the specified subarray, the algorithm is repeated on one-quarter of the original array. The search continues until the search key is equal to a subarray's middle element, or until the subarray consists of one element that's not equal to the search key (i.e., the search key is not found).
- When using a binary search, the maximum number of comparisons required for any array can be determined by finding the first power of 2 greater than the number of array elements.

Section 6.11 Multidimensional Arrays

- A common use of multidimensional arrays (p. 278) is to represent tables of values consisting of rows and columns. To identify a particular table element, we specify two subscripts. By convention, the first identifies the element's row, and the second identifies its column.
- Arrays that require two subscripts to identify an element are called **two-dimensional arrays** (p. 278). Multidimensional arrays can have more than two subscripts.
- A multidimensional array can be initialized when it's defined. The values in a two-dimensional array's initializer list are grouped by row in braces. If there are not enough initializers for a given row, its remaining elements are initialized to 0.
- The first subscript of a multidimensional array parameter declaration is not required, but all subsequent subscripts are. The compiler uses these sizes to determine the locations in memory of elements in multidimensional arrays. All array elements are stored consecutively in memory, regardless of the number of subscripts. In a two-dimensional array, the first row is stored in memory, followed by the second row, and so on.
- Providing the subscript values in a parameter declaration enables the compiler to tell the function how to locate an array element. In a two-dimensional array, each row is basically a one-dimensional array. To locate an element in a particular row, the compiler must know how many elements are in each row, so it can skip the proper number of memory locations when accessing elements of a given row.

Section 6.12 Variable-Length Arrays

- A variable-length array (p. 286) is an array for which the size is defined by an expression evaluated at execution time.
- When applied to a variable-length array, **sizeof** operates at runtime.
- Variable-length arrays are optional in C—they may not be supported by your compiler.

Self-Review Exercises

6.1	Answer each of the following:
	a) Lists and tables of values are stored in
	b) The number used to refer to a particular array element is called its
	c) A(n) should be used to specify the size of an array because it makes
	the program more modifiable.
	d) Placing the elements of an array in order is called the array.
	e) Determining whether an array contains a key value is called the array
	f) An array that uses two subscripts is referred to as a(n) array

- **6.2** State whether the following are *true* or *false*. If the answer is *false*, explain why.
 - a) An array can store many different types of values.
 - b) An array subscript can be of data type double.
 - c) If there are fewer initializers in an initializer list than there are array elements, the remaining elements are initialized with the initializer list's last value.
 - d) It's an error if an initializer list contains more initializers than there are array elements.
 - e) An individual array element that's passed to a function as an argument of the form a[i] and modified in the called function will contain the modified value in the calling function.
- **6.3** Follow the instructions below regarding an array called fractions.
 - a) Define a symbolic constant SIZE with the replacement text 10.
 - b) Define a double array with SIZE elements and initialize the elements to 0.
 - c) Refer to array element 4.
 - d) Assign the value 1.667 to array element nine.
 - e) Assign the value 3.333 to the seventh element of the array.
 - f) Print array elements 6 and 9 with two digits of precision to the right of the decimal point, and show the output that's displayed on the screen.
 - g) Print all the elements of an array using a for iteration statement. Use the variable x as the loop's control variable. Show the output.
- **6.4** Write statements to accomplish the following:
 - a) Define table to be an integer array and to have 3 rows and 3 columns. Assume the symbolic constant SIZE has been defined to be 3.
 - b) How many elements does the array table contain? Print the total number of elements.
 - c) Use a for iteration statement to initialize each element of table to the sum of its subscripts. Use variables x and y as control variables.
 - d) Print the values of each element of array table. Assume the array was initialized with the definition:

```
int table[SIZE][SIZE] = \{\{1, 8\}, \{2, 4, 6\}, \{5\}\}\};
```

6.5 Find the error in each of the following program segments and correct the error.

```
a) #define SIZE 100;
b) SIZE = 10;
c) int b[10] = {0};
  int i;
  for (size_t i = 0; i <= 10; ++i) {
     b[i] = 1;
  }
d) #include <stdio.h>;
e) int a[2][2] = {{1, 2}, {3, 4}};
  a[1, 1] = 5;
f) #define VALUE = 120
```

Answers to Self-Review Exercises

- **6.1** a) arrays. b) subscript (or index). c) symbolic constant. d) sorting. e) searching. f) two-dimensional.
- **6.2** a) *False*. An array can store only values of the same type.
 - b) False. An array subscript must be an integer or an integer expression.
 - c) False. C automatically initializes the remaining elements to zero.
 - d) True.
 - e) *False*. Individual elements of an array are passed by value. If the entire array is passed to a function, any modifications will be reflected in the original array.

```
6.3
     a) #define SIZE 10
     b) double fractions[SIZE] = {0.0};
     c) fractions[4]
     d) fractions[9] = 1.667;
     e) fractions[6] = 3.333;
     f) printf("%.2f %.2f\n", fractions[6], fractions[9]);
        Output: 3.33 1.67.
     g) for (size_t x = 0; x < SIZE; ++x) {
           printf("fractions[%zu] = %f\n", x, fractions[x]);
        }
      Output:
          fractions[0] = 0.000000
          fractions[1] = 0.000000
          fractions[2] = 0.000000
          fractions[3] = 0.000000
          fractions[4] = 0.000000
          fractions[5] = 0.000000
          fractions[6] = 3.333000
          fractions[7] = 0.000000
          fractions[8] = 0.000000
          fractions[9] = 1.667000
6.4 a) int table[SIZE][SIZE];
     b) Nine elements. printf("%d\n", SIZE * SIZE);
     c) for (size_t x = 0; x < SIZE; ++x) {
           for (size_t y = 0; y < SIZE; ++y) {
              table[x][y] = x + y;
           }
        }
     d) for (size_t x = 0; x < SIZE; ++x) {
           for (size_t y = 0; y < SIZE; ++y) {
              printf("table[%d][%d] = %d\n", x, y, table[x][y]);
           }
        }
      Output:
          table[0][0] = 1
```

table[0][1] = 8 table[0][2] = 0 table[1][0] = 2 table[1][1] = 4 table[1][2] = 6 table[2][0] = 5 table[2][1] = 0 table[2][2] = 0

- 6.5 a) Error: Semicolon at the end of the #define preprocessor directive. Correction: Eliminate semicolon.
 - b) Error: Assigning a value to a symbolic constant using an assignment statement. Correction: Assign a value to the symbolic constant in a #define preprocessor directive without using the assignment operator, as in #define SIZE 10.
 - c) Error: Referencing an array element outside the bounds of the array (b[10]). Correction: Change the control variable's final value to 9 or change <= to <.
 - d) Error: Semicolon at the end of the #include preprocessor directive. Correction: Eliminate semicolon.
 - e) Error: The array subscripting is done incorrectly.

 Correction: Change the statement to a[1][1] = 5;
 - f) Error: A symbolic constant's value is not defined using =.

 Correction: Change the preprocessor directive to #define VALUE 120.

Exercises

6.6	Fill in the blanks in each of the following:
	a) C stores lists of values in
	b) An array's elements are related by the fact that they
	c) When referring to an array element, the position number contained within
	square brackets is called a(n)
	d) The names of array p's five elements are,,
	and
	e) The content of a particular array element is called that element's
	f) Naming an array, stating its type and specifying its number of elements is
	called the array.
	g) Placing an array's elements into either ascending or descending order is
	called
	h) In a two-dimensional array, the first subscript identifies the element's
	and the second identifies its
	i) An <i>m</i> -by- <i>n</i> array contains rows, columns and el-
	ements.
	j) The name of the element in row 3 and column 5 of array d is
<i>c</i> 7	State will be Called Called and American and American and Called To Called and Line

- **6.7** State which of the following are *true* and which are *false*. If *false*, explain why.
 - a) To refer to a particular location or element within an array, specify the array's name and the value of the particular element.
 - b) An array definition reserves space for the array.

- c) To indicate that 100 locations should be reserved for integer array p, write p[100];
- d) A program that initializes a 15-element array's elements to zeros must contain a for statement.
- e) A program that totals the elements of a two-dimensional array must contain nested for statements.
- f) The mean, median and mode of the following set of values are 5, 6 and 7, respectively: 1, 2, 5, 6, 7, 7, 7.
- **6.8** Write statements to accomplish each of the following:
 - a) Display the value of the seventh element of character array f.
 - b) Input a value into element 4 of one-dimensional floating-point array b.
 - c) Initialize each of the five elements of one-dimensional integer array g to 8.
 - d) Total the elements of floating-point 100-element array c.
 - e) Copy array a into the first portion of array b. Assume a has 11 elements, be has 34 elements, and both arrays have the same element type.
 - f) Determine and print the smallest and largest values contained in 99-element floating-point array w.
- **6.9** Consider a 2-by-5 integer array t.
 - a) Write a definition for t.
 - b) How many rows does t have?
 - c) How many columns does t have?
 - d) How many elements does t have?
 - e) Write the names of all the elements in the second row of t.
 - f) Write the names of all the elements in the third column of t.
 - g) Write a single statement that sets the element of t in row 1 and column 2 to 0.
 - h) Write a series of statements that initialize each element of t to zero. Do not use an iteration statement.
 - i) Write a nested for statement that initializes each element of t to zero.
 - j) Write a statement that inputs the values for the elements of t from the terminal.
 - k) Write a series of statements that determine and print the smallest value in array t.
 - l) Write a statement that displays the elements of the first row of t.
 - m)Write a statement that totals the elements of the fourth column of t.
 - n) Write a series of statements that print the array t in tabular format. List the column subscripts as headings across the top and list the row subscripts at the left of each row.
- **6.10** (Sales Commissions) Use a one-dimensional array to solve the following problem. A company pays its salespeople on a commission basis. The salespeople receive \$200 per week plus 9% of their gross sales for that week. For example, a salesperson who grosses \$3000 in sales in a week receives \$200 plus 9% of \$3000 for a total of \$470. Write a C program (using an array of counters) that determines how many

salespeople earned salaries in each of the following ranges—assume that each salesperson's salary is truncated to an integer amount:

- a) \$200–299
- b) \$300-399
- c) \$400-499
- d) \$500-599
- e) \$600-699
- f) \$700-799
- g) \$800-899
- h) \$900–999
- i) \$1000 and over
- **6.11** (*Bubble Sort*) The bubble sort presented in Fig. 6.12 is inefficient for large arrays. Make the following modifications to improve its performance:
 - a) After the first pass, the largest number is guaranteed to be in the highestnumbered array element; after the second pass, the two highest numbers are "in place," and so on. Instead of making nine comparisons on every pass, modify the bubble sort to make eight comparisons on the second pass, seven on the third pass and so on.
 - b) The data in the array may already be in the proper or near-proper order, so why make nine passes if fewer will suffice? Modify the sort to check at the end of each pass whether any swaps have been made. If there were none, then the data must already be in the proper order, so the sort should terminate. If swaps have been made, then at least one more pass is needed.
- **6.12** Write loops that perform each of the following one-dimensional array operations:
 - a) Initialize the 10 elements of integer array counts to zeros.
 - b) Add 1 to each of the 15 elements of integer array bonus.
 - c) Read the 12 values of float array monthlyTemperatures from the keyboard.
 - d) Print the five values of integer array bestScores in column format.
- **6.13** Find the error(s) in each of the following statements:

```
a) Assume: char str[5] = "";
    scanf("%s", str); // User types hello
b) Assume: int a[3];
    printf("$d %d %d\n", a[1], a[2], a[3]);
c) double f[3] = {1.1, 10.01, 100.001, 1000.0001};
d) Assume: double d[2][10] = {0};
    d[1, 9] = 2.345;
```

6.14 (Mean, Median and Mode Program Modifications) Modify the program of Fig. 6.13 so function mode can handle a tie for the mode value. If there are two values with the same frequency, the data is "bimodal" and both values should be displayed. If there are more than two values with the same frequency, the data is "multimodal" and all the values with the same frequency should be displayed. Also modify the me-

dian function to average the two middle elements in an array with an even number of elements.

- **6.15** (*Duplicate Elimination*) Use a one-dimensional array to solve the following problem. Read 20 numbers, each of which is between 10 and 100, inclusive. As each number is read, print it only if it's not a duplicate of a number already read. Provide for the "worst case" in which all 20 numbers are different. Use the smallest possible array to solve this problem.
- **6.16** Label the elements of 3-by-5 two-dimensional array sales to indicate the order in which they're set to zero by the following program segment:

```
for (size_t row = 0; row <= 2; ++row) {
   for (size_t column = 0; column <= 4; ++column) {
      sales[row][column] = 0;
   }
}</pre>
```

6.17 What does the following program do?

```
// ex06_17.c
 2 // What does this program do?
 3 #include <stdio.h>
 4 #define SIZE 10
 5
   int whatIsThis(const int b[], size_t p); // function prototype
 7
 8
   int main(void) {
 9
       // initialize array a
       int a[SIZE] = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\};
10
11
12
       int x = whatIsThis(a, SIZE);
13
       printf("Result is %d\n", x);
    }
14
15
16
    // what does this function do?
17
   int whatIsThis(const int b[], size_t p) {
       if (1 == p) { // base case
18
19
          return b[0];
20
21
       else { // recursion step
          return b[p-1] + whatIsThis(b, p-1);
22
23
       }
24
    }
```

6.18 What does the following program do?

```
// ex06_18.c
// what does this program do?
#include <stdio.h>
#define SIZE 10
// function prototype
void someFunction(const int b[], size_t start, size_t size);
```

```
8
 9
    // function main begins program execution
10
    int main(void) {
       int a[SIZE] = \{8, 3, 1, 2, 6, 0, 9, 7, 4, 5\}; // initialize a
11
12
13
       puts("Answer is:");
       someFunction(a, 0, SIZE);
14
       puts("");
15
16
    }
17
    // What does this function do?
    void someFunction(const int b[], size_t start, size_t size) {
19
20
       if (start < size) {</pre>
21
           someFunction(b, start + 1, size);
          printf("%d ", b[start]);
22
23
       }
24
    }
```

6.19 (*Dice Rolling*) Write a program that simulates rolling two dice. The program should use rand twice to roll the first and second die, respectively, then calculate their sum. Because each die can have an integer value from 1 to 6, the sum of the values will vary from 2 to 12, with 7 being the most frequent sum and 2 and 12 the least frequent sums. The following diagram shows the 36 possible combinations of the two dice:

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Your program should roll the two dice 36,000 times. Use a one-dimensional array to tally the numbers of times each possible sum appears. Print the results in tabular format. Also, determine whether the totals are reasonable—for example, there are six ways to roll a 7, so approximately one-sixth of all the rolls should be 7.

- **6.20** (*Craps Game Statistics*) Write a program that runs 1,000,000 games of craps (without human intervention) and answers each of the following questions:
 - a) How many games are won on the first roll, second roll, ..., twentieth roll and after the twentieth roll?
 - b) How many games are lost on the first roll, second roll, ..., twentieth roll and after the twentieth roll?
 - c) What are the chances of winning at craps? You should discover that craps is one of the fairest casino games. What do you suppose this means?
 - d) What's the average length of a game of craps?
 - e) Do the chances of winning improve with the length of the game?

6.21 (Airline Reservations System) A small airline has just purchased a computer for its new automated reservations system. The president has asked you to program the new system. You'll write a program to assign seats on each flight of the airline's only plane (capacity: 10 seats).

Your program should display the following menu of alternatives:

```
Please type 1 for "first class" Please type 2 for "economy"
```

If the person types 1, assign a seat in the first-class section (seats 1–5). If the person types 2, assign a seat in the economy section (seats 6–10). Your program should then print a boarding pass indicating the person's seat number and whether it's in the first-class or economy section of the plane.

Use a one-dimensional array to represent the plane's seating chart. Initialize all the elements of the array to 0 to indicate that all seats are empty. As each seat is assigned, set the corresponding element of the array to 1 to indicate that the seat is no longer available.

Your program should, of course, never assign a seat that has already been assigned. When the first-class section is full, your program should ask the person if it's acceptable to be placed in the economy section (and vice versa). If yes, then make the appropriate seat assignment. If no, then print the message, "Next flight leaves in 3 hours."

- **6.22** (*Total Sales*) Use a two-dimensional array to solve the following problem. A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each different type of product sold. Each slip contains:
 - a) The salesperson number
 - b) The product number
 - c) The total dollar value of that product sold that day

Thus, each salesperson passes in between 0 and 5 sales slips per day. Assume that the information from all of the slips for last month is available. Write a program that reads all this sales information and summarizes the total sales by salesperson by product. All totals should be stored in the two-dimensional array sales. After processing all the information for last month, print the results in tabular format with each column representing a particular salesperson and each row representing a particular product. Cross-total each row to get the total sales of each product for last month; cross total each column to get the total sales by salesperson for last month. Your tabular printout should include these cross totals to the right of the totaled rows and to the bottom of the totaled columns.

6.23 (*Turtle Graphics*) The Logo language made the concept of *turtle graphics* famous. Imagine a mechanical turtle that walks around the room under the control of a C program. The turtle holds a pen in one of two positions, up or down. While the pen is down, the turtle traces out shapes as it moves; while the pen is up, the turtle moves about freely without writing anything. In this problem, you'll simulate the operation of the turtle and create a computerized sketchpad as well.

Use a 50-by-50 array floor that's initialized to zeros. Read commands from an array that contains them. Keep track of the current turtle position at all times and whether the pen is currently up or down. Assume that the turtle always starts at position 0, 0 of the floor with its pen up. The set of turtle commands your program must process are shown in the following table:

Command	Meaning
1	Pen up
2	Pen down
3	Turn right
4	Turn left
5, 10	Move forward 10 spaces (or a number other than 10)
6	Print the 50-by-50 array
9	End of data (sentinel)

Suppose that the turtle is somewhere near the center of the floor. The following "program" would draw and print a 12-by-12 square:

2 5,12 3 5,12 3 5,12 3 5,12 1 6

As the turtle moves with the pen down, set elements of array floor to 1s. When the 6 command is given, display an asterisk for each 1 in the array. For each zero, display a blank. Write a program to implement the turtle-graphics capabilities discussed here. Write several turtle graphics programs to draw interesting shapes. Add other commands to increase the power of your turtle-graphics language.

6.24 (*Knight's Tour*) One of the more interesting puzzles for chess buffs is the Knight's Tour problem, originally proposed by the mathematician Euler. The question is this: Can the chess piece called the knight move around an empty chessboard and touch each of the 64 squares once and only once? We study this intriguing problem in-depth here.

The knight makes L-shaped moves (two in one direction and then one in a perpendicular direction). Thus, from a square in the middle of an empty chessboard, the knight can make eight different moves (numbered 0 through 7), as shown in the following diagram:

	0	1	2	3	4	5	6	7
0								
1				2		1		
2			3				0	
3					K			
4			4				7	
5				5		6		
6								
7								

- a) Draw an 8-by-8 chessboard on a sheet of paper and attempt a Knight's Tour by hand. Put a 1 in the first square you move to, a 2 in the second square, a 3 in the third, and so on. Before starting the tour, estimate how far you think you'll get, remembering that a full tour consists of 64 moves. How far did you get? Were you close to the estimate?
- b) Now let's develop a program that will move the knight around a chessboard. The board itself is represented by an 8-by-8 two-dimensional array board. Each square is initialized to zero. We describe each of the eight possible moves in terms of both its horizontal and vertical components. For example, a move of type 0, as shown in the preceding diagram, consists of moving two squares horizontally to the right and one square vertically upward. Move 2 consists of moving one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two one-dimensional arrays, horizontal and vertical, as follows:

```
horizontal[0] = 2
                       vertical[0] = -1
horizontal[1] = 1
                       vertical[1] = -2
horizontal[2] = -1
                       vertical[2] = -2
horizontal[3] = -2
                       vertical[3] = -1
horizontal[4] = -2
                       vertical[4] = 1
horizontal[5] = -1
                       vertical[5] = 2
horizontal[6] = 1
                       vertical[6] = 2
horizontal[7] = 2
                       vertical[7] = 1
```

The variables currentRow and currentColumn indicate the row and column of the knight's current position. To make a move of type moveNumber, where moveNumber is between 0 and 7, your program uses the statements

```
currentRow += vertical[moveNumber];
currentColumn += horizontal[moveNumber];
```

Keep a counter that varies from 1 to 64. Record the latest count in each square the knight moves to. Remember to test each potential move to see if the knight has already visited that square. And, of course, test every potential move to make sure that the knight does not land off the chessboard.

- Now write a program to move the knight around the chessboard. Run the program. How many moves did the knight make?
- c) After attempting to write and run a Knight's Tour program, you have probably developed some valuable insights. We'll use these to develop a *heuristic* (or strategy) for moving the knight. Heuristics do not guarantee success, but a carefully developed heuristic greatly improves the chance of success. You may have observed that the outer squares are in some sense more trouble-some than the squares nearer the center of the board. In fact, the most troublesome, or inaccessible, squares are the four corners.

Intuition may suggest that you should attempt to move the knight to the most troublesome squares first and leave open those that are easiest to get to, so that when the board gets congested near the end of the tour, there will be a greater chance of success.

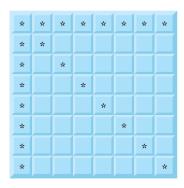
We develop an "accessibility heuristic" by classifying each square according to how accessible it is and always moving the knight to the square (within the knight's L-shaped moves, of course) that's most inaccessible. We label a two-dimensional array accessibility with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, the center squares are therefore rated as 8s, the corner squares are rated as 2s, and the other squares have accessibility numbers of 3, 4, or 6 as follows:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

Now write a version of the Knight's Tour program using the accessibility heuristic. At any time, the knight should move to the square with the lowest accessibility number. In case of a tie, the knight may move to any of the tied squares. Therefore, the tour may begin in any of the four corners. [Note: As the knight moves around the chessboard, your program should reduce the accessibility numbers as more and more squares become occupied. In this way, at any given time during the tour, each available square's accessibility number will remain equal to precisely the number of squares from which that square may be reached.] Run this version of your program. Did you get a full tour? (Optional: Modify the program to run 64 tours, one from each square of the chessboard. How many full tours did you get?)

d) Write a version of the Knight's Tour program which, when encountering a tie between two or more squares, decides what square to choose by looking ahead to those squares reachable from the "tied" squares. Your program should move to the square for which the next move would arrive at a square with the lowest accessibility number.

- **6.25** (Knight's Tour: Brute-Force Approaches) In Exercise 6.24, we developed a Knight's Tour solution. The approach used, called the "accessibility heuristic," generates many solutions and executes efficiently. As computers continue increasing in power, we'll be able to solve many problems with sheer computer power and relatively unsophisticated algorithms. Let's call this approach brute-force problem solving.
 - a) Use random numbers to enable the knight to walk at random around the chessboard in its legitimate L-shaped moves. Your program should run one tour and print the final chessboard. How far did the knight get?
 - b) Most likely, the preceding program produced a relatively short tour. Now modify your program to attempt 1000 tours. Use a one-dimensional array to keep track of the number of tours of each length. When your program finishes attempting the 1000 tours, it should print this information in a tabular format. What was the best result?
 - c) Most likely, the preceding program gave you some "respectable" tours but no full tours. Now "pull all the stops out" and simply let your program run until it produces a full tour. [Caution: This could run for hours on a powerful computer.] Once again, track the number of tours of each length and print this table when the first full tour is found. How many tours did your program attempt before producing a full tour? How much time did it take?
 - d) Compare the Knight's Tour brute-force version with the accessibility-heuristic version. Which required more careful study of the problem? Which was more difficult to develop? Which required more computer power? Could we be certain (in advance) of obtaining a full tour with the accessibility heuristic? Could we be certain (in advance) of obtaining a full tour with brute force? Argue the pros and cons of brute-force problem solving in general.
- **6.26** (*Eight Queens*) Another puzzle for chess buffs is the Eight Queens problem. Simply stated: Is it possible to place eight queens on an empty chessboard so that no queen is "attacking" any other—that is, so that no two queens are in the same row, the same column, or along the same diagonal? Use the kind of thinking developed in Exercise 6.24 to formulate a heuristic for solving the Eight Queens problem. Run your program. *Hint:* It's possible to assign a numeric value to each square of the chessboard indicating how many squares of an empty chessboard are "eliminated" once a queen is placed in that square. For example, each of the four corners would be assigned the value 22, as illustrated in the following diagram:



Once these "elimination numbers" are placed in all 64 squares, an appropriate heuristic might be: Place the next queen in the square with the smallest elimination number. Why is this strategy intuitively appealing?

- **6.27** (*Eight Queens: Brute-Force Approaches*) In this exercise, you'll develop several brute-force approaches to solving the Eight Queens problem introduced in Exercise 6.26.
 - a) Solve the Eight Queens problem, using the random brute-force technique developed in Exercise 6.25.
 - b) Use an exhaustive technique (i.e., try all possible combinations of eight queens on the chessboard).
 - c) Why do you suppose the exhaustive brute-force approach may not be appropriate for solving the Eight Queens problem?
 - d) Compare and contrast the random brute-force and exhaustive brute-force approaches in general.
- **6.28** (*Duplicate Elimination*) In Chapter 12, we explore the high-speed binary search tree data structure. One feature of a binary search tree is that duplicate values are discarded when insertions are made into the tree. This is referred to as duplicate elimination. Write a program that produces 20 random numbers between 1 and 20. The program should store all nonduplicate values in an array. Use the smallest possible array to accomplish this task.
- **6.29** (Knight's Tour: Closed Tour Test) In the Knight's Tour, a full tour occurs when the knight makes 64 moves touching each square of the chessboard once and only once. A closed tour occurs when the 64th move is one move away from the location in which the knight started the tour. Modify the Knight's Tour program you wrote in Exercise 6.24 to test for a closed tour if a full tour has occurred.
- **6.30** (*The Sieve of Eratosthenes*) A prime integer is any integer greater than 1 that can be divided evenly only by itself and 1. In this exercise, you'll use the Sieve of Eratosthenes to find all the prime numbers less than 1000. It works as follows:
 - a) Create a 100-element array with all elements initialized to 1 (true). Array elements with prime subscripts will remain 1. All other array elements will eventually be set to zero.
 - b) Starting with subscript 2 (1 is not prime), every time an array element is found whose value is 1, loop through the remainder of the array and set to zero every element whose subscript is a multiple of the subscript for the element with value 1. For array subscript 2, all elements beyond 2 in the array that are multiples of 2 will be set to zero (subscripts 4, 6, 8, 10, and so on). For array subscript 3, all elements beyond 3 in the array that are multiples of 3 will be set to zero (subscripts 6, 9, 12, 15, and so on).

When this process is complete, the array elements that are still set to 1 indicate that the subscript is a prime number. Write a program that determines and prints the prime numbers between 1 and 999. Ignore element 0 of the array.

Recursion Exercises

- **6.31** (*Palindromes*) A palindrome is a string that's spelled the same way forward and backward. Some examples of palindromes are: "radar," "able was i ere i saw elba," and, if you ignore blanks, "a man a plan a canal panama." Write a recursive function testPalindrome that returns 1 if the string stored in the array is a palindrome and 0 otherwise. The function should ignore spaces and punctuation in the string.
- **6.32** (*Linear Search*) Modify the program of Fig. 6.14 to use a recursive linear-Search function to perform the linear search of the array. The function should receive an integer array, the size of the array and the search key as arguments. If the search key is found, return the array subscript; otherwise, return –1.
- **6.33** (Binary Search) Modify the program of Fig. 6.15 to use a recursive binarySearch function to perform the binary search of the array. The function should receive an integer array, the starting subscript, the ending subscript and the search key as arguments. If the search key is found, return the array subscript; otherwise, return -1.
- **6.34** (*Eight Queens*) Modify the Eight Queens program you created in Exercise 6.26 to solve the problem recursively.
- **6.35** (*Print an Array*) Write a recursive function printArray that takes an array and the size of the array as arguments, prints the array, and returns nothing. The function should stop processing and return when it receives an array of size zero.
- **6.36** (*Print a String Backward*) Write a recursive function stringReverse that takes a character array as an argument, prints it back-to-front and returns nothing. The function should stop processing and return when the terminating null character of the string is encountered.
- **6.37** (Find the Minimum Value in an Array) Write a recursive function recursive-Minimum that takes an integer array and the array size as arguments and returns the smallest element of the array. The function should stop processing and return when it receives an array of one element.

Pointers



7

Objectives

In this chapter, you'll:

- Use pointers and pointer operators.
- Pass arguments to functions by reference using pointers.
- Understand the const qualifier's various placements and how they affect what operations you can perform on a variable.
- Use the **sizeof** operator with variables and types.
- Use pointer arithmetic to process array elements.
- Understand the close relationships among pointers, arrays and strings.
- Define and use arrays of strings.
- Use function pointers.
- Learn about secure C programming with pointers.

- **7.1** Introduction
- **7.2** Pointer Variable Definitions and Initialization
- **7.3** Pointer Operators
- **7.4** Passing Arguments to Functions by Reference
- **7.5** Using the **const** Qualifier with Pointers
 - 7.5.1 Converting a String to Uppercase Using a Non-Constant Pointer to Non-Constant Data
 - 7.5.2 Printing a String One Character at a Time Using a Non-Constant Pointer to Constant Data
 - 7.5.3 Attempting to Modify a Constant Pointer to Non-Constant Data
 - 7.5.4 Attempting to Modify a Constant Pointer to Constant Data
- **7.6** Bubble Sort Using Pass-By-Reference
- **7.7 sizeof** Operator
- **7.8** Pointer Expressions and Pointer Arithmetic
 - 7.8.1 Pointer Arithmetic Operators
 - 7.8.2 Aiming a Pointer at an Array
 - 7.8.3 Adding an Integer to a Pointer

- 7.8.4 Subtracting an Integer from a Pointer
- 7.8.5 Incrementing and Decrementing a Pointer
- 7.8.6 Subtracting One Pointer from Another
- 7.8.7 Assigning Pointers to One Another
- 7.8.8 Pointer to void
- 7.8.9 Comparing Pointers
- **7.9** Relationship between Pointers and Arrays
 - 7.9.1 Pointer/Offset Notation
 - 7.9.2 Pointer/Subscript Notation
 - 7.9.3 Cannot Modify an Array Name with Pointer Arithmetic
 - 7.9.4 Demonstrating Pointer Subscripting and Offsets
 - 7.9.5 String Copying with Arrays and Pointers
- **7.10** Arrays of Pointers
- **7.11** Random-Number Simulation Case Study: Card Shuffling and Dealing
- **7.12** Function Pointers
 - 7.12.1 Sorting in Ascending or Descending Order
 - 7.12.2 Using Function Pointers to Create a Menu-Driven System
- **7.13** Secure C Programming

Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises | Array of Function Pointer Exercises | Special Section: Building Your Own Computer as a Virtual Machine | Special Section; Embedded Systems Programming Case Study: Robotics with the Webots Simulator

7.1 Introduction

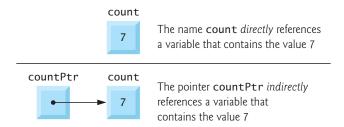
In this chapter, we discuss one of C's most powerful features—the **pointer**. Pointers enable programs to

- accomplish pass-by-reference,
- pass functions between functions,
- manipulate strings and arrays, and
- create and manipulate dynamic data structures that grow and shrink at execution time, such as linked lists, queues, stacks and trees.

This chapter explains basic pointer concepts. In Section 7.13, we discuss various pointer-related security issues. Chapter 10 examines using pointers with structures. Chapter 12 introduces dynamic memory management and shows how to create and use dynamic data structures.

7.2 Pointer Variable Definitions and Initialization

Pointers are variables whose values are memory addresses. Usually, a variable *directly* contains a specific value. A pointer, however, contains the address of another variable that contains a specific value. The pointer *points to* that variable. In this sense, a variable name directly references a value, and a pointer indirectly references a value, as in the following diagram:



Referencing a value through a pointer is called indirection.

Declaring Pointers

Pointers, like all variables, must be defined before they can be used. The following statement defines the variable countPtr as an int *—a pointer to an integer:

```
int *countPtr;
```

This definition is read right-to-left, "countPtr is a pointer to int" or "countPtr points to an object¹ of type int." The * indicates that the variable is a pointer.

Pointer Variable Naming

Our convention is to end each pointer variable's name with Ptr to indicate that the variable is a pointer and should be handled accordingly. Other common naming conventions include starting the variable name with p (e.g., pCount) or p_ (e.g., p_count).

Define Variables in Separate Statements

The * in the following definition does not distribute to each variable:

```
int *countPtr, count;
```

so countPtr is a pointer to int, but count is just an int. For this reason, you should always write the preceding declaration as two statements to prevent ambiguity:

```
int *countPtr;
int count;
```

Initializing and Assigning Values to Pointers

Pointers should be initialized when they're defined, or they can be assigned a value. A pointer may be initialized to NULL, 0 or an address:

^{1.} In C, an "object" is a region of memory that can hold a value. So objects in C include primitive types such as ints, floats, chars and doubles, as well as aggregate types such as arrays and structs (which we discuss in Chapter 10).

- A pointer with the value NULL points to *nothing*. NULL is a *symbolic constant* with the value 0 and is defined in the header <stddef.h> (and several other headers, such as <stdio.h>).
- Initializing a pointer to 0 is equivalent to initializing it to NULL. The constant NULL is preferred because it emphasizes that you're initializing a pointer rather than a variable that stores a number. When 0 is assigned, it's first converted to a pointer of the appropriate type. The value 0 is the only integer value that can be assigned directly to a pointer variable.



Assigning a variable's address to a pointer is discussed in Section 7.3. Initialize
pointers to prevent unexpected results.

✓ Self Check

I (*True*/*False*) The definition:

```
int *countPtr, count;
```

specifies that countPtr and count are of type int *—each is a pointer to an integer. Answer: *False*. Actually, count is an int, not a pointer to an int. The * applies only to countPtr and does not distribute to the other variable(s) in the definition.

- **2** (*Multiple Choice*) Which of the following statements is *false*?
 - a) A pointer may be initialized to NULL, 0 or an address.
 - b) Initializing a pointer to 0 is equivalent to initializing a pointer to NULL, but 0 is preferred.
 - c) The only integer that can be assigned directly to a pointer variable is 0.
 - d) Initialize pointers to prevent unexpected results.

Answer: b) is *false*. Actually, NULL is preferred because it highlights the fact that the variable is of a pointer type.

7.3 Pointer Operators

Next, let's discuss the address (&) and indirection (*) operators, and their relationship.

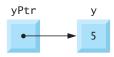
The Address (&) Operator

The unary address operator (&) returns the *address* of its operand. For example, given the following definition of y:

int
$$y = 5$$
;

the statement

initializes pointer variable yPtr with variable y's *address*—yPtr is then said to "point to" y. The following diagram shows the variables yPtr and y in memory:



Pointer Representation in Memory

The following diagram shows the preceding pointer's representation in memory, assuming that integer variable y is stored at location 600000 and the pointer variable yPtr is stored at location 500000:



The operand of & must be a variable; the address operator *cannot* be applied to literal values (like 27 or 41.5) or expressions.

The Indirection (*) Operator

You apply the unary indirection operator (*), also called the dereferencing operator, to a pointer operand to get the *value* of the object to which the pointer points. For example, the following statement prints 5, which is the value of variable y:

```
printf("%d", *yPtr);
```

Using * in this manner is called dereferencing a pointer.

Dereferencing a pointer that has not been initialized with or assigned the address \bigotimes ERR of another variable in memory is an error. This could

- cause a fatal execution-time error,
- accidentally modify important data and allow the program to run to completion with incorrect results, or
- lead to a security breach.²

Demonstrating the & and * Operators

Figure 7.1 demonstrates the pointer operators & and *. The printf conversion specification %p outputs a memory location as a hexadecimal integer on most platforms.³ The output shows that the *address* of a and the *value* of aPtr are identical, confirming that a's address was indeed assigned to the pointer variable aPtr (line 7). The & and * operators are complements of one another. Applying both consecutively to aPtr in either order (line 12) produces the same result. The addresses in the output will vary across systems that use different processor architectures, different compilers and even different compiler settings.

```
// fig07_01.c
// Using the & and * pointer operators.
#include <stdio.h>
int main(void) {
```

Fig. 7.1 | Using the & and * pointer operators. (Part 1 of 2.)

^{2.} https://cwe.mitre.org/data/definitions/824.html.

^{3.} See online Appendix E for more information on hexadecimal integers.

```
6
       int a = 7;
7
       int *aPtr = &a; // set aPtr to the address of a
8
9
       printf("Address of a is %p\nValue of aPtr is %p\n\n", &a, aPtr);
       printf("Value of a is %d\nValue of *aPtr is %d\n\n", a, *aPtr);
10
       printf("Showing that * and & are complements of each other\n");
II
       printf("\&*aPtr = %p\n*\&aPtr = %p\n", &*aPtr, *\&aPtr);
12
13
Address of a is 0x7fffe69386cc
Value of aPtr is 0x7fffe69386cc
Value of a is 7
Value of *aPtr is 7
Showing that * and & are complements of each other
&*aPtr = 0x7fffe69386cc
*&aPtr = 0x7fffe69386cc
```

Fig. 7.1 Using the & and * pointer operators. (Part 2 of 2.)

The following table lists the precedence and grouping of the operators introduced to this point:

Operators	Grouping	Туре
() [] ++ (postfix) (postfix)	left to right	postfix
+ - ++ ! * & (type)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > S>=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
H	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Self Check

I (*True/False*) Assuming the definitions

```
double d = 98.6;
double *dPtr;
```

the following statement assigns variable d's address to the pointer variable dPtr:

```
dPtr = &d;
```

Variable dPtr is then said to "point to" d.

Answer: True.

2 (*Fill-In*) The unary indirection operator (*) returns the value of the object to which its pointer operand points. Using * in this manner is called _____. Answer: dereferencing a pointer.

7.4 Passing Arguments to Functions by Reference

There are two ways to pass arguments to a function—pass-by-value and pass-by-reference. By default, arguments (other than arrays) are passed by value. As you've seen, arrays are passed by reference. Functions often need to modify variables in the caller or to receive a pointer to a large data object to avoid the overhead of copying the object (as in pass-by-value). As we saw in Chapter 5, a return statement can return at most one value from a called function to its caller. Pass-by-reference also can enable a function to "return" multiple values by modifying the caller's variables.

Use & and * to Accomplish Pass-By-Reference

Pointers and the indirection operator enable pass-by-reference. When calling a function with arguments that should be modified in the caller, you use & to pass each variable's address. As we saw in Chapter 6, arrays are *not* passed using operator & because an array's name is equivalent to & arrayName[0]—the array's starting location in memory. A function that receives the address of a variable in the caller can use the indirection operator (*) to modify the value at that location in the caller's memory, thus effecting pass-by-reference.

Pass-By-Value

The programs in Figs. 7.2 and 7.3 present two versions of a function that cubes an integer—cubeByValue and cubeByReference. Line 11 of Fig. 7.2 passes the variable number by value to function cubeByValue (lines 16–18), which cubes its argument and returns the new value. Line 11 assigns the new value to number in main, replacing number's value.

```
// fig07_02.c
   // Cube a variable using pass-by-value.
2
3
   #include <stdio.h>
4
5
   int cubeByValue(int n); // prototype
7
    int main(void) {
       int number = 5; // initialize number
8
9
       printf("The original value of number is %d", number);
10
       number = cubeByValue(number); // pass number by value to cubeByValue
11
       printf("\nThe new value of number is %d\n", number);
12
13
    }
14
```

Fig. 7.2 Cube a variable using pass-by-value. (Part 1 of 2.)

```
15  // calculate and return cube of integer argument
16  int cubeByValue(int n) {
17    return n * n * n; // cube local variable n and return result
18  }

The original value of number is 5
The new value of number is 125
```

Fig. 7.2 Cube a variable using pass-by-value. (Part 2 of 2.)

Pass-By-Reference

Line 12 of Fig. 7.3 passes the variable number's address to function cubeByReference (lines 17–19)—passing the address enables pass-by-reference. The function's parameter is a pointer to an int called nPtr (line 17). The function uses the expression *nPtr to dereference the pointer and cube the value to which it points (line 18). It assigns the result to *nPtr—which is really the variable number in main—thus changing number's value in main. Use pass-by-value unless the caller explicitly requires the called function to modify the argument variable's value in the caller. This prevents accidental modification of the caller's arguments and is another example of the principle of least privilege.

```
I // fig07_03.c
2 // Cube a variable using pass-by-reference with a pointer argument.
4
  #include <stdio.h>
5
6
   void cubeByReference(int *nPtr); // function prototype
7
8
   int main(void) {
9
       int number = 5; // initialize number
10
       printf("The original value of number is %d", number);
11
12
       cubeByReference(&number); // pass address of number to cubeByReference
13
       printf("\nThe new value of number is %d\n", number);
   }
14
15
16 // calculate cube of *nPtr; actually modifies number in main
17 void cubeByReference(int *nPtr) {
       *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
18
19
The original value of number is 5
The new value of number is 125
```

Fig. 7.3 Cube a variable using pass-by-reference with a pointer argument.

Use a Pointer Parameter to Receive an Address

A function receiving an address as an argument must receive it in a pointer parameter. For example, in Fig. 7.3, function cubeByReference's header (line 17) is

```
void cubeByReference(int *nPtr) {
```

which specifies that cubeByReference receives the address of an integer variable as an argument, stores the address locally in parameter nPtr and does not return a value.

Pointer Parameters in Function Prototypes

The function prototype for cubeByReference (Fig. 7.3, line 6) specifies an int * parameter. As with other parameters, it's not necessary to include pointer names in function prototypes—they're ignored by the compiler—but it's good practice to include them for documentation purposes.

Functions That Receive One-Dimensional Arrays

For a function that expects a one-dimensional array argument, the function's prototype and header can use the pointer notation shown in the parameter list of function cubeByReference (line 17). The compiler does not differentiate between a function that receives a pointer and one that receives a one-dimensional array. So, the function must "know" when it's receiving an array vs. a single variable passed by reference. When the compiler encounters a function parameter for a one-dimensional array of the form int b[], the compiler converts the parameter to the pointer notation int *b. The two forms are interchangeable. Similarly, for a parameter of the form const int b[] the compiler converts the parameter to const int *b.

Pass-By-Value vs. Pass-By-Reference Step-By-Step

Figures 7.4 and 7.5 analyze graphically and step-by-step the programs in Figs. 7.2 and 7.3, respectively.

```
Step 1: Before main calls cubeByValue:
 int main(void) {
                                        number
    int number = 5;
                                          5
    number = cubeByValue(number);
Step 2: After cubeByValue receives the call:
 int main(void) {
                                        number
                                                     int cubeByValue( int n ) {
    int number = 5;
                                                        return n * n * n;
                                                                                     n
    number = cubeByValue(number);
                                                     }
                                                                                     5
Step 3: After cubeByValue cubes parameter n and before cubeByValue returns to main:
 int main(void) {
                                                     int cubeByValue(int n) {
                                        number
                                                                    125
                                           5
    int number = 5;
                                                        return n * n * n;
                                                                                     5
    number = cubeByValue(number);
                                                     }
```

Fig. 7.4 | Analysis of a typical pass-by-value. (Part 1 of 2.)

Step 4: After cubeByValue returns to main and before assigning the result to number:

```
int main(void) {
   int number = 5;
   number = cubeByValue(number);
}
```

Step 5: After main completes the assignment to number:

```
int main(void) {
   int number = 5;
   125
   number = cubeByValue(number);
}
```

Fig. 7.4 | Analysis of a typical pass-by-value. (Part 2 of 2.)

Step 1: Before main calls cubeByReference:

```
int main(void)
{
   int number = 5;
   cubeByReference(&number);
}
```

Step 2: After cubeByReference receives the call and before *nPtr is cubed:

```
int main(void)
{
  int number = 5;
  cubeByReference(&number);
}

void cubeByReference(int *nPtr)
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
    nPtr

call establishes this pointer
```

Step 3: After *nPtr is cubed and before program control returns to main:

```
int main(void)
{
  int number = 5;
  cubeByReference(&number);
}

void cubeByReference(int *nPtr)
{
    125
    *nPtr = *nPtr * *nPtr * *nPtr;
}
  called function modifies caller's
    variable
```

Fig. 7.5 Analysis of a typical pass-by-reference with a pointer argument.

Self Check

- (Multiple Choice) Which of the following statements is false?
 - a) By default, arguments (other than arrays) are passed by value. Arrays are passed by reference.
 - b) Functions often require the capability to modify variables in the caller or receive a pointer to a large data object to avoid copying the object.
 - c) return can return one or more values from a called function to a caller.
 - d) Pass-by-reference also can enable a function to "return" multiple values by modifying variables in the caller.

Answer: c) is *false*. return may be used to return at most one value from a called function to a caller.

- (Multiple Choice) Which of the following statements is false?
 - a) You use pointers and the indirection operator to accomplish pass-by-refer-
 - b) When calling a function with arguments that should be modified, use the address operator (&) to pass the argument's addresses.
 - c) Arrays are passed by reference using operator &.
 - d) All of the above statements are *true*.

Answer: c) is false. Arrays are not passed by reference using operator & because an array's name is equivalent to the address of its first element—& arrayName[0].

7.5 Using the const Qualifier with Pointers

The const qualifier enables you to inform the compiler that a particular variable's value should not be modified, thus enforcing the principle of least privilege. This can ASE reduce debugging time and prevent unintentional side effects, making a program more robust, and easier to modify and maintain. If an attempt is made to modify a 💢 ERR value that's declared const, the compiler catches it and issues an error.





Over the years, a large base of legacy code was written in early C versions that did not use const because it was not available. Even more current code does not use const as often as it should. So, there are significant opportunities for improvement by reengineering existing C code.

There are four ways to pass to a function a pointer to data:

- a non-constant pointer to non-constant data.
- a constant pointer to non-constant data.
- a non-constant pointer to constant data.
- a constant pointer to constant data.

Each of the four combinations provides different access privileges and is discussed in the next several examples. How do you choose one of the possibilities? Let the principle of least privilege be your guide. Always award a function enough access to the data in its parameters to accomplish its specified task, but absolutely no more.

7.5.1 Converting a String to Uppercase Using a Non-Constant Pointer to Non-Constant Data

The highest level of data access is granted by a non-constant pointer to non-constant data. The data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data items. A function might use such a pointer to receive a string argument, then process (and possibly modify) each character in the string. Function convertToUppercase in Fig. 7.6 declares its parameter, a non-constant pointer to non-constant data called sPtr (line 18). The function processes the array string (pointed to by sPtr) one character at a time. C standard library function toupper (line 20) from the <ctype.h> header converts each character to its corresponding uppercase letter. If the original character is not a letter or is already uppercase, toupper returns the original character. Line 21 increments the pointer to point to the next character in the string. Chapter 8 presents many C standard library character- and string-processing functions.

```
// fig07_06.c
 2 // Converting a string to uppercase using a
 3 // non-constant pointer to non-constant data.
   #include <ctype.h>
 5
   #include <stdio.h>
 6
 7
   void convertToUppercase(char *sPtr); // prototype
 8
 9
   int main(void) {
       char string[] = "cHaRaCters and $32.98"; // initialize char array
10
11
       printf("The string before conversion is: %s\n", string);
12
13
       convertToUppercase(string);
       printf("The string after conversion is: %s\n", string);
14
15
    }
16
17
   // convert string to uppercase letters
    void convertToUppercase(char *sPtr) {
18
       while (*sPtr != ) { // current character is not
19
20
          *sPtr = toupper(*sPtr); // convert to uppercase
21
          ++sPtr; // make sPtr point to the next character
22
       }
23
    }
The string before conversion is: cHaRaCters and $32.98
The string after conversion is: CHARACTERS AND $32.98
```

Fig. 7.6 | Converting a string to uppercase using a non-constant pointer to non-constant data.

7.5.2 Printing a String One Character at a Time Using a Non-Constant Pointer to Constant Data

A non-constant pointer to constant data can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified. A function

might receive such a pointer to process an array argument's elements without modifying them. For example, function printCharacters (Fig. 7.7) declares parameter sPtr to be of type const char * (line 20). The declaration is read from *right to left* as "sPtr is a pointer to a character constant." The function's for statement outputs each character until it encounters a null character. After displaying each character, the loop increments pointer sPtr to point to the string's next character.

```
I // fig07_07.c
 2 // Printing a string one character at a time using
 3 // a non-constant pointer to constant data.
 4
 5
   #include <stdio.h>
 6
 7
   void printCharacters(const char *sPtr);
 8
 9 int main(void) {
10
       // initialize char array
11
       char string[] = "print characters of a string";
12
13
       puts("The string is:");
14
       printCharacters(string);
15
       puts("");
16
   }
17
18 // sPtr cannot be used to modify the character to which it points,
19 // i.e., sPtr is a "read-only" pointer
20 void printCharacters(const char *sPtr) {
21
       // loop through entire string
22
       for (; *sPtr != ; ++sPtr) { // no initialization
         printf("%c", *sPtr);
23
24
25
    }
The string is:
print characters of a string
```

Fig. 7.7 Printing a string one character at a time using a non-constant pointer to constant data.

Trying to Modify Constant Data

Figure 7.8 shows the errors from compiling a function that receives a non-constant pointer (xPtr) to constant data and tries to use it to modify the data. The error shown is from the Visual C++ compiler. The C standard does not specify compiler warning or error messages, and the compiler vendors do not normalize these messages across compilers. So, the actual error message you receive is compiler-specific. For example, Xcode's LLVM compiler reports the error:

```
error: read-only variable is not assignable and the GNU gcc compiler reports the error: error: assignment of read-only location '*xPtr'
```

```
// fig07_08.c
   // Attempting to modify data through a
3 // non-constant pointer to constant data.
4 #include <stdio.h>
   void f(const int *xPtr); // prototype
5
7
    int main(void) {
8
       int y = 7; // define y
9
10
       f(&y); // f attempts illegal modification
11
   }
12
   // xPtr cannot be used to modify the
13
   // value of the variable to which it points
15 void f(const int *xPtr) {
       *xPtr = 100; // error: cannot modify a const object
16
17
Microsoft Visual C++ Error Message
fig07_08.c(16,5): error C2166: 1-value specifies const object
```

Attempting to modify data through a non-constant pointer to constant data.

Passing Structures vs. Arrays

As you know, arrays are aggregate types that store related data items of the same type under one name. Chapter 10 discusses another form of aggregate type called a structure (sometimes called a record or tuple in other languages), which can store related data items of the same or different types under one name—e.g., employee information, such as an employee's ID number, name, address and salary.

Unlike arrays, structures are passed by value—a copy of the entire structure is passed. This requires the execution-time overhead of making a copy of each data item PERF : in the structure and storing it on the computer's function call stack. Passing large objects such as structures by using pointers to constant data obtains the performance of pass-by-reference and the security of pass-by-value. In this case, the program copies only the address at which the structure is stored—typically four or eight bytes.

> If memory is low and execution efficiency is a concern, use pointers. If memory is in abundance and efficiency is not a major concern, pass data by value to enforce the principle of least privilege. Some systems do not enforce const well, so pass-byvalue is still the best way to prevent data from being modified.

7.5.3 Attempting to Modify a Constant Pointer to Non-Constant Data

A constant pointer to non-constant data always points to the *same* memory location, but the data at that location can be modified through the pointer. This is the default for an array name, which is a constant pointer to the array's first element. All data in the array can be accessed and changed by using the array name and array subscripting. A constant pointer to non-constant data can be used to receive an array as an argu-

ment to a function that accesses array elements using array subscript notation. Pointers that are declared const must be initialized when they're defined. If the pointer is a function parameter, it's initialized with a pointer argument as the function is called.

Figure 7.9 attempts to modify a constant pointer. Pointer ptr is defined in line 11 to be of type int * const, which is read *right-to-left* as "ptr is a constant pointer to an integer." The pointer is initialized (line 11) with the address of integer variable x. The program attempts to assign y's address to ptr (line 14), but the compiler generates an error.

```
// fig07_09.c
   // Attempting to modify a constant pointer to non-constant data.
3 #include <stdio.h>
4
5
   int main(void) {
6
       int x = 0; // define x
7
       int y = 0; // define y
8
       // ptr is a constant pointer to an integer that can be modified
9
       // through ptr, but ptr always points to the same memory location
10
       int * const ptr = &x;
П
12
       *ptr = 7; // allowed: *ptr is not const
13
       ptr = &y; // error: ptr is const; cannot assign new address
14
    }
15
Microsoft Visual C++ Error Message
fig07_09.c(14,4): error C2166: 1-value specifies const object
```

Fig. 7.9 | Attempting to modify a constant pointer to non-constant data.

7.5.4 Attempting to Modify a Constant Pointer to Constant Data

The *least* access privilege is granted by a **constant pointer to constant data**. Such a pointer always points to the *same* memory location, and the data at that memory location *cannot be modified*. This is how an array should be passed to a function that only looks at the array's elements using array subscript notation and does *not* modify the elements. Figure 7.10 defines pointer variable ptr (line 12) to be of type const int *const, which is read *right-to-left* as "ptr is a constant pointer to an integer constant." The output shows the error messages generated when we attempt to *modify the data* to which ptr points (line 15) and when we attempt to *modify* the *address* stored in the pointer variable (line 16).

```
// fig07_10.c
// Attempting to modify a constant pointer to constant data.
#include <stdio.h>
```

Fig. 7.10 | Attempting to modify a constant pointer to constant data. (Part 1 of 2.)

```
5
   int main(void) {
6
       int x = 5;
7
       int y = 0;
8
9
       // ptr is a constant pointer to a constant integer. ptr always
       // points to the same location; the integer at that location
10
       // cannot be modified
П
       const int *const ptr = &x; // initialization is OK
12
13
       printf("%d\n", *ptr);
14
15
       *ptr = 7; // error: *ptr is const; cannot assign new value
       ptr = &y; // error: ptr is const; cannot assign new address
16
17
   }
```

Microsoft Visual C++ Error Message

```
fig07_10.c(15,5): error C2166: l-value specifies const object fig07_10.c(16,4): error C2166: l-value specifies const object
```

Fig. 7.10 Attempting to modify a constant pointer to constant data. (Part 2 of 2.)

Self Check

[(Multiple Choice) What is sPtr in the following prototype?

void convertToUppercase(char *sPtr);

- a) A non-constant pointer to constant data.
- b) A constant pointer to non-constant data.
- c) A non-constant pointer to non-constant data.
- d) A constant pointer to constant data.

Answer: c.

2 (*Fill-In*) The least access privilege is granted by a _____ pointer to _____ data. Such a pointer always points to the same memory location, and the data at that memory location cannot be modified.

Answer: constant, constant.

7.6 Bubble Sort Using Pass-By-Reference

Let's improve the bubble-sort⁴ program of Fig. 6.12 to use two functions—bubble-Sort and swap (Fig. 7.11). Function bubbleSort sorts the array. It calls function swap (line 42) to exchange the array elements array[j] and array[j + 1].

^{4.} In Chapter 12 and Appendix C, we investigate sorting schemes that yield better performance.

```
// fig07_11.c
    // Putting values into an array, sorting the values into
 3 // ascending order and printing the resulting array.
   #include <stdio.h>
 4
 5
    #define SIZE 10
 6
 7
    void bubbleSort(int * const array, size_t size); // prototype
9
    int main(void) {
10
       // initialize array a
       int a[SIZE] = \{ 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 \};
11
12
13
       puts("Data items in original order");
14
15
       // loop through array a
16
       for (size_t i = 0; i < SIZE; ++i) {
          printf("%4d", a[i]);
17
18
       }
19
20
       bubbleSort(a, SIZE); // sort the array
21
22
       puts("\nData items in ascending order");
23
24
       // loop through array a
25
       for (size_t i = 0; i < SIZE; ++i) {
          printf("%4d", a[i]);
26
27
28
29
       puts("");
30
    }
31
32
    // sort an array of integers using bubble sort algorithm
    void bubbleSort(int * const array, size_t size) {
       void swap(int *element1Ptr, int *element2Ptr); // prototype
34
35
36
       // loop to control passes
37
       for (int pass = 0; pass < size - 1; ++pass) {
38
          // loop to control comparisons during each pass
39
          for (size_t j = 0; j < size - 1; ++j) {
             // swap adjacent elements if they're out of order
40
41
             if (array[j] > array[j + 1]) {
42
                swap(\&array[j], \&array[j + 1]);
43
44
          }
       }
45
46 }
47
```

Fig. 7.11 Putting values into an array, sorting the values into ascending order and printing the resulting array. (Part 1 of 2.)

```
// swap values at memory locations to which element1Ptr and
// element2Ptr point
void swap(int *element1Ptr, int *element2Ptr) {
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
}

Data items in original order
   2 6 4 8 10 12 89 68 45 37
Data items in ascending order
   2 4 6 8 10 12 37 45 68 89
```

Fig. 7.11 Putting values into an array, sorting the values into ascending order and printing the resulting array. (Part 2 of 2.)

Function swap

Remember that C enforces *information hiding* between functions, so swap does not have access to individual array elements in bubbleSort by default. Because bubbleSort wants swap to have access to the array elements to swap, bubbleSort passes each element's address to swap, so the elements are passed by reference. Although entire arrays are automatically passed by reference, individual array elements are *scalars* and are ordinarily passed by value. So, bubbleSort uses the address operator (&) on each array element:

```
swap(\&array[j], \&array[j + 1]);
```

Function swap receives &array[j] in element1Ptr (line 50). Function swap may use *element1Ptr as a synonym for array[j]. Similarly, *element2Ptr is a synonym for array[j + 1]. Even though swap is not allowed to say

```
int hold = array[j];
  array[j] = array[j + 1];
  array[j + 1] = hold;

precisely the same effect is achieved by lines 51 through 53:
  int hold = *element1Ptr;
  *element1Ptr = *element2Ptr:
```

Function bubbleSort's Array Parameter

*element2Ptr = hold;

Note that function bubbleSort's header (line 33) declares array as int * const array rather than int array[] to indicate that bubbleSort receives a one-dimensional array argument. Again, these notations are interchangeable; however, array notation generally is preferred for readability.

Function swap's Prototype in Function bubbleSort's Body

The prototype for function swap (line 34) is included in bubbleSort's body because only bubbleSort calls swap. Placing the prototype in bubbleSort restricts proper swap

calls to those made from bubbleSort (or any function that appears after swap in the source code). Other functions defined before swap that attempt to call swap do not have access to a proper function prototype, so the compiler generates one automatically. This normally results in a prototype that does not match the function header (and generates a compilation warning or error) because the compiler assumes int for the return and parameter types. Placing function prototypes in the definitions of ASE other functions enforces the principle of least privilege by restricting proper function calls to the functions in which the prototypes appear.

Function bubbleSort's size Parameter

Function bubbleSort receives the array size as a parameter (line 33). When an array is passed to a function, the memory address of the array's first element, of course, does not convey the number of array elements. Therefore, you must pass the array size to the function to know how many elements to sort. Another common practice is to pass a pointer to the array's first element and a pointer to the location just beyond the array's end. As you'll learn in Section 7.8, the difference between these two pointers is the array's length, and the resulting code is simpler.

There are two main benefits to passing the array size to bubbleSort—software 📥 SE reusability and proper software engineering. By defining the function to receive the array size as an argument, we enable the function to be used by any program that sorts one-dimensional integer arrays of any size.

We could have stored the array's size in a global variable accessible to the entire program. However, other programs that require an integer array-sorting capability may not have the same global variable, so the function cannot be used in those programs. Global variables usually violate the principle of least privilege and can lead to 📥 SE poor software engineering. Global variables should be used only to represent truly shared resources, such as the time of day.

The array size could have been programmed directly into the function. This would restrict the function's use to processing an array of a specific size and significantly reduce its reusability. Only programs processing one-dimensional integer arrays of the specific size coded into the function can use the function.

Self Check

(Code) Our bubbleSort function used the address operator (&) on each of the array elements in the swap call to effect pass-by-reference as follows:

```
swap(\&array[j], \&array[j + 1]);
```

Suppose function swap receives & array[j] and & array[j + 1] in int * pointers named firstPtr and secondPtr, respectively. Write the pointer-based code in function swap to switch the values in these two elements, using a temporary int variable temp.

Answer:

```
int temp = *firstPtr;
*firstPtr = *secondPtr;
*secondPtr = temp;
```

2 (*Discussion*) Typically, when we pass an array to a function, we also pass the array size as another argument. Alternatively, we could build the array size directly into the function definition. What's wrong with that approach?

Answer: It would limit the function to processing arrays of a specific size, significantly reducing the function's reusability.

7.7 sizeof Operator

C provides the unary operator **sizeof** to determine an object's or type's size in bytes. This operator is applied at compilation time unless its operand is a variable-length array (VLA; Section 6.12). When applied to an array's name as in Fig. 7.12 (line 12), sizeof returns as a size_t value the array's total number of bytes. Variables of type float on our computer are stored in four bytes of memory, and array is defined to have 20 elements. Therefore, there are 80 bytes in array. sizeof is a compile-time operator, so it does not incur any execution-time overhead (except for VLAs).

```
I // fig07_12.c
2 // Applying sizeof to an array name returns
3 // the number of bytes in the array.
4 #include <stdio.h>
5 #define SIZE 20
7
   size_t getSize(const float *ptr); // prototype
8
9
   int main(void){
10
       float array[SIZE]; // create array
11
       printf("Number of bytes in the array is %zu\n", sizeof(array));
12
13
       printf("Number of bytes returned by getSize is %zu\n", getSize(array));
14
   }
15
   // return size of ptr
16
   size_t getSize(const float *ptr) {
17
       return sizeof(ptr);
18
19
   }
Number of bytes in the array is 80
Number of bytes returned by getSize is 8
```

Fig. 7.12 Applying **sizeof** to an array name returns the number of bytes in the array.

Even though function getSize receives an array of 20 elements as an argument, the function's parameter ptr is simply a pointer to the array's first element. When you use sizeof with a pointer, it returns the *pointer's size*, not the size of the item to which it points. On our 64-bit Windows, Mac and Linux test systems, a pointer's size is eight bytes, so getSize returns 8. On older 32-bit systems, a pointer's size is typically four bytes, so getSize would return 4.

The number of elements in an array also can be determined with sizeof. For example, consider the following array definition:

```
double real[22];
```

Variables of type double normally are stored in eight bytes of memory. Thus, the array real contains 176 bytes. The following expression determines the array's number of elements:

```
sizeof(real) / sizeof(real[0])
```

The expression divides the array real's number of bytes by the number of bytes used to store one element of the array (a double value). This calculation works *only* when using the actual array's name, *not* when using a pointer to the array.

Determining the Sizes of the Standard Types, an Array and a Pointer

Figure 7.13 calculates the number of bytes used to store each of the standard types. The results of this program are implementation dependent. They often differ across platforms and sometimes across different compilers on the same platform. The output shows the results from our Mac system using the Xcode C++ compiler.

```
I // fig07_13.c
 2 // Using operator sizeof to determine standard type sizes.
 3 #include <stdio.h>
 5 int main(void) {
       char c = ;
 6
 7
       short s = 0;
 8
       int i = 0;
 9
       long 1 = 0;
10
       long long 11 = 0;
       float f = 0.0F;
11
12
       double d = 0.0;
13
       long double ld = 0.0;
       int array[20] = {0}; // create array of 20 int elements
14
15
       int *ptr = array; // create pointer to array
16
                   sizeof c = %2zu\t
                                            sizeof(char) = \%2zu\n''
17
       printf("
          sizeof c, sizeof(char));
18
19
       printf("
                  sizeof s = %2zu\t
                                           sizeof(short) = %2zu\n",
          sizeof s, sizeof(short));
20
       printf("
                                             sizeof(int) = %2zu\n",
21
                 sizeof i = %2zu\t
          sizeof i, sizeof(int));
22
23
       printf("
                 sizeof 1 = %2zu\t
                                            sizeof(long) = %2zu\n'',
24
          sizeof 1, sizeof(long));
       printf(" sizeof 11 = %2zu\t sizeof(long long) = %2zu\n",
25
26
          sizeof 11, sizeof(long long));
27
       printf("
                   sizeof f = %2zu\t
                                           sizeof(float) = %2zu\n",
          sizeof f, sizeof(float));
28
29
                 sizeof d = %2zu\t
                                          sizeof(double) = %2zu\n",
       printf("
          sizeof d, sizeof(double));
30
```

Fig. 7.13 Using operator sizeof to determine standard type sizes. (Part 1 of 2.)

```
sizeof ld = %2zu\tsizeof(long double) = %2zu\n",
31
          sizeof ld, sizeof(long double));
32
       printf("sizeof array = %2zu\n sizeof ptr = %2zu\n",
33
          sizeof array, sizeof ptr);
35 }
    sizeof c = 1
                             sizeof(char) = 1
    sizeof s = 2
                            sizeof(short) = 2
    sizeof i = 4
                              sizeof(int) = 4
    sizeof 1 = 8
                             sizeof(long) = 8
   sizeof 11 = 8
                       sizeof(long long) = 8
                           sizeof(float) = 4
    sizeof f = 4
    sizeof d = 8
                           sizeof(double) = 8
   sizeof ld = 16
                      sizeof(long double) = 16
sizeof array = 80
  sizeof ptr = 8
```

Fig. 7.13 Using operator **sizeof** to determine standard type sizes. (Part 2 of 2.)



The number of bytes used to store a particular type may vary between systems. When writing programs that depend on type sizes and that will run on several computer systems, use sizeof to determine the number of bytes used to store the types.

You can apply sizeof to any variable name, type or value (including the value of an expression). When applied to a variable name (that's *not* an array name) or a constant, the number of bytes used to store the specific type of variable or constant is returned. The parentheses are required when a type is supplied as sizeof's operand.

Self Check

I (*Fill-In*) Given the array definition:

```
double temperatures[31];
the expression:
    sizeof(temperatures) / sizeof(temperatures[0])
determines what attribute of temperatures? _____
Answer: The number of elements in the array (in this case, 31).
```

2 (*TruelFalse*) When you use sizeof with a pointer, it returns the size of the item to which the pointer points.

Answer: False. Actually, when you use sizeof with a pointer, it returns the pointer's size, not the size of the item to which the pointer points. If you use sizeof with an array name, it returns the array's size.

7.8 Pointer Expressions and Pointer Arithmetic

Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions. However, not all arithmetic operators are valid with pointer variables. This section describes the operators that can have pointers as operands, and how these operators are used.

7.8.1 Pointer Arithmetic Operators

The following arithmetic operations are allowed for pointers:

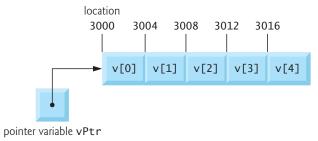
- incrementing (++) or decrementing (--),
- adding an integer to a pointer (+ or +=),
- subtracting an integer from a pointer (- or -=), and
- subtracting one pointer from another—meaningful only when *both* pointers point into the *same* array.

Pointer arithmetic on pointers that do not refer to array elements is a logic error.



7.8.2 Aiming a Pointer at an Array

Assume the array int v[5] is defined, and its first element is at location 3000 in memory. Also, assume the pointer vPtr points to v[0]—so the value of vPtr is 3000. The following diagram illustrates this scenario for a machine with four-byte integers:



The variable vPtr can be initialized to point to array v with either of the statements

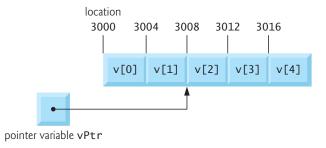
```
vPtr = v;
vPtr = &v[0];
```

7.8.3 Adding an Integer to a Pointer

In conventional arithmetic, 3000 + 2 yields the value 3002. This is normally not the case with pointer arithmetic. When you add an integer to or subtract one from a pointer, the pointer increments or decrements by that integer *times the size of the object to which the pointer refers*. For example, the statement

$$vPtr += 2;$$

would produce 3008 (3000 + 2 * 4), assuming an integer is stored in four bytes of memory. In the array v, vPtr would now point to v[2], as in the following diagram:



The object's size, depends on its type. When performing pointer arithmetic on a character array, the results will be consistent with regular arithmetic because each character is one byte. Type sizes can vary by platform and compiler, so pointer arithmetic is platform- and compiler-dependent.

7.8.4 Subtracting an Integer from a Pointer

If vPtr had been incremented to 3016 (v[4]), the statement

```
vPtr -= 4;
```



would set vPtr back to 3000 (v[0])—the beginning of the array. Using pointer arith-ERR (x) metic to adjust pointers to point outside an array's bounds is a logic error that could SEC lead to security problems.

7.8.5 Incrementing and Decrementing a Pointer

To increment or decrement a pointer by one, use the increment (++) and decrement (--) operators. Either of the statements

```
++vPtr;
vPtr++;
```

increments the pointer to point to the next array element. Either of the statements

```
--vPtr:
vPtr--;
```

decrements the pointer to point to the previous array element.

7.8.6 Subtracting One Pointer from Another

If vPtr contains the location 3000 and v2Ptr contains the address 3008, the statement

```
x = v2Ptr - vPtr:
```



assigns to x the *number of array elements* between vPtr and v2Ptr, in this case, 2 (not ERR (x) 8). Pointer arithmetic is undefined unless performed on elements of the same array. We cannot assume that two variables of the same type are stored side-by-side in memory unless they're adjacent elements of an array.

7.8.7 Assigning Pointers to One Another

Pointers of the same type may be assigned to one another. This rule's exception is a pointer to void (i.e., void *)—a generic pointer that can represent any pointer type. All pointer types can be assigned to a void *, and a void * can be assigned a pointer of any type (including another void *). In both cases, a cast operation is *not* required.

7.8.8 Pointer to void

A pointer to void *cannot* be dereferenced. Consider this: The compiler knows on a machine with four-byte integers that an int * points to four bytes of memory. However, a void * contains a memory location for an *unknown* type—the precise number of bytes to which the pointer refers is *not* known by the compiler. The compiler *must* know the type to determine the number of bytes that represent the referenced value. Dereferencing a void * pointer is a syntax error.



7.8.9 Comparing Pointers

You can compare pointers using equality and relational operators, but such comparisons are meaningful only if the pointers point to elements of the same array; otherwise, such comparisons are logic errors. Pointer comparisons compare the addresses (X) ERR stored in the pointers. Such a comparison could show, for example, that one pointer points to a higher-numbered array element than the other. A common use of pointer comparison is determining whether a pointer is NULL.





Self Check

- [(Fill-In) When you add an integer to or subtract an integer from a pointer, the pointer increments or decrements by that integer times _ **Answer:** the size of the object to which the pointer points.
- **2** (Fill-In) Pointers v1Ptr and v2Ptr point to elements of the same array of eightbyte double values. If v1Ptr contains the address 3000 and v2Ptr contains the address 3016, then the statement

```
size_t x = v2Ptr - v1Ptr;
will assign _____ to x.
Answer: 2 (not 16)—2 is the number of elements between the pointers.
```

7.9 Relationship between Pointers and Arrays

Arrays and pointers are intimately related and often may be used interchangeably. You can think of an array name as a constant pointer to the array's first element. Pointers can be used to do any operation involving array subscripting.

Assume the following definitions:

```
int b[5];
int *bPtr;
```

Because the array name b (without a subscript) is a pointer to the array's first element, we can set bPtr to the address of the array b's first element with the statement:

```
bPtr = b:
```

This is equivalent to taking the address of array b's first element as follows:

```
bPtr = &b[0]:
```

7.9.1 Pointer/Offset Notation

Array element b[3] can alternatively be referenced with the pointer expression

```
*(bPtr + 3)
```

The 3 in the expression is the offset to the pointer. When bPtr points to the array's first element, the offset indicates which array element to reference—the offset's value

is identical to the array subscript. This notation is referred to as **pointer/offset notation**. The parentheses are required because the precedence of * is *higher* than that of +. Without the parentheses, the above expression would add 3 to the value of the expression *bPtr (i.e., 3 would be added to b[0], assuming bPtr points to the beginning of the array). Just as the array element can be referenced with a pointer expression, the address

```
&b[3]
```

can be written with the pointer expression

```
bPtr + 3
```

An array's name also can be treated as a pointer and used in pointer arithmetic. For example, the expression

```
*(b + 3)
```

refers to element b[3]. In general, all subscripted array expressions can be written with a pointer and an offset. In this case, pointer/offset notation was used with the array's name as a pointer. The preceding statement does not modify the array name in any way; b still points to the first element.

7.9.2 Pointer/Subscript Notation

Pointers can be subscripted like arrays. If bPtr has the value b, the expression

```
bPtr[1]
```

refers to the array element b[1]. This is referred to as pointer/subscript notation.

7.9.3 Cannot Modify an Array Name with Pointer Arithmetic

An array name always points to the beginning of the array, so it's like a constant pointer. Thus, the expression

```
b += 3
```

is *invalid* because it attempts to modify the array name's value with pointer arithme-ERR \bigotimes tic. Attempting to modify the value of an array name with pointer arithmetic is a compilation error.

7.9.4 Demonstrating Pointer Subscripting and Offsets

Figure 7.14 uses the four methods we've discussed for referring to array elements—array subscripting, pointer/offset with the array name as a pointer, pointer subscripting, and pointer/offset with a pointer—to print the four elements of the integer array b.

```
I // fig07_14.cpp
```

Fig. 7.14 Using subscripting and pointer notations with arrays. (Part 1 of 3.)

^{2 //} Using subscripting and pointer notations with arrays.

^{3 #}include <stdio.h>

^{4 #}define ARRAY SIZE 4

```
5
 6
    int main(void) {
 7
       int b[] = \{10, 20, 30, 40\}; // create and initialize array b
8
       int *bPtr = b; // create bPtr and point it to array b
 9
10
       // output array b using array subscript notation
11
       puts("Array b printed with:\nArray subscript notation");
12
13
       // loop through array b
14
       for (size_t i = 0; i < ARRAY_SIZE; ++i) {
15
          printf("b[%zu] = %d\n", i, b[i]);
       }
16
17
18
       // output array b using array name and pointer/offset notation
19
       puts("\nPointer/offset notation where the pointer is the array name");
20
21
       // loop through array b
22
       for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {</pre>
23
          printf("*(b + %zu) = %d\n", offset, *(b + offset));
24
25
26
       // output array b using bPtr and array subscript notation
27
       puts("\nPointer subscript notation");
28
29
       // loop through array b
       for (size_t i = 0; i < ARRAY_SIZE; ++i) {</pre>
30
31
          printf("bPtr[%zu] = %d\n", i, bPtr[i]);
32
33
34
       // output array b using bPtr and pointer/offset notation
35
       puts("\nPointer/offset notation");
36
37
       // loop through array b
       for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {</pre>
38
39
          printf("*(bPtr + %zu) = %d\n", offset, *(bPtr + offset));
40
       }
41 }
Array b printed with:
Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40
Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40
```

Fig. 7.14 Using subscripting and pointer notations with arrays. (Part 2 of 3.)

```
Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Fig. 7.14 Using subscripting and pointer notations with arrays. (Part 3 of 3.)

7.9.5 String Copying with Arrays and Pointers

To further illustrate array and pointer interchangeability, let's look at two string-copying functions—copy1 and copy2—in Fig. 7.15. Both functions copy a string into a character array, but they're implemented differently.

```
I // fig07_15.c
 2 // Copying a string using array notation and pointer notation.
 3 #include <stdio.h>
 4 #define SIZE 10
 5
 6 void copy1(char * const s1, const char * const s2); // prototype
 7
   void copy2(char *s1, const char *s2); // prototype
 8
9
   int main(void) {
       char string1[SIZE]; // create array string1
10
       char *string2 = "Hello"; // create a pointer to a string
11
12
13
       copy1(string1, string2);
14
       printf("string1 = %s\n", string1);
15
16
       char string3[SIZE]; // create array string3
17
       char string4[] = "Good Bye"; // create an array containing a string
18
19
       copy2(string3, string4);
20
       printf("string3 = %s\n", string3);
21
   }
22
23
   // copy s2 to s1 using array notation
   void copy1(char * const s1, const char * const s2) {
24
25
       // loop through strings
       for (size_t i = 0; (s1[i] = s2[i]) != ; ++i) {
26
27
          ; // do nothing in body
28
29
    }
```

Fig. 7.15 Copying a string using array notation and pointer notation. (Part 1 of 2.)

```
30
    // copy s2 to s1 using pointer notation
31
void copy2(char *s1, const char *s2) {
       // loop through strings
       for (; (*s1 = *s2) != ; ++s1, ++s2) {
34
          ; // do nothing in body
35
36
    }
37
string1 = Hello
string3 = Good Bye
```

Copying a string using array notation and pointer notation. (Part 2 of 2.)

Copying with Array Subscript Notation

Function copy1 uses array subscript notation to copy the string in s2 to the character array s1. The function defines counter variable i as the array subscript. The for statement header (line 26) performs the entire copy operation. The statement's body is the empty statement. The header specifies that i is initialized to zero and incremented by one during each iteration. The expression s1[i] = s2[i] copies one character from s2 to s1. When the null character is encountered in s2, it's assigned to s1. Since the assignment's value is what gets assigned to the left operand (\$1), the loop terminates when an element of s1 receives the null character, which has the value 0 and therefore is false.

Copying with Pointers and Pointer Arithmetic

Function copy2 uses pointers and pointer arithmetic to copy the string in s2 to the character array s1. Again, the for statement header (line 34) performs the copy operation. The header does not include any variable initialization. The expression *s1 = *s2 performs the copy operation by dereferencing s2 and assigning that character to the current location in \$1. After the assignment, line 34 increments \$1 and \$2 to point to each string's next character. When the assignment copies the null character into s1, the loop terminates.

Notes Regarding Functions copy1 and copy2

The first argument to both copy1 and copy2 must be an array large enough to hold the second argument's string. Otherwise, a logic error may occur when an attempt is made to (X) ERR write into a memory location that's not part of the array. In both functions, the second argument is copied into the first argument—characters are read from it one at a time, but the characters are never modified. Therefore, the second parameter is declared to point to a constant value so that the *principle of least privilege* is enforced. Neither function requires the capability of modifying the string in the second argument, so we simply disallow it.

Self Check

- I (*True/False*) If bPtr points to array b's second element (b[1]), then element b[3] also can be referenced with the pointer/offset notation expression *(bPtr + 3). Answer: *False*. Since the pointer points to array b's second element (b[1]), the expression should be *(bPtr + 2).
- **2** (*Fill-In*) Pointers can be subscripted like arrays. If bPtr points to the array b's first element, the expression

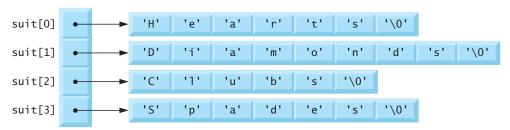
```
bPtr[1]
refers to the array element _____.
Answer: b[1].
```

7.10 Arrays of Pointers

Arrays may contain pointers. A common use of an array of pointers is to form an array of strings, referred to simply as a string array. Each element in a C string is essentially a pointer to its first character. So, each entry in an array of strings is actually a pointer to a string's first character. Consider the definition of the string array suit, which might be useful in representing a deck of cards.

```
const char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
```

The array has four elements. The char * indicates that each suit element is of type "pointer to char." The qualifier const indicates that the string each element points to cannot be modified. The strings "Hearts", "Diamonds", "Clubs" and "Spades" are placed into the array. Each is stored in memory as a *null-terminated character string* that's one character longer than the number of characters in the quotes. So, the strings are 7, 9, 6 and 7 characters long. Although it appears these strings are being placed into the array, only pointers are actually stored, as shown in the following diagram:



Each pointer points to the first character of its corresponding string. Thus, even though a char * array is *fixed* in size, it can point to character strings of *any length*. This flexibility is one example of C's powerful data-structuring capabilities.

The suits could have been placed in a two-dimensional array, in which each row would represent a suit, and each column would represent a letter from a suit name. Such a data structure would have to have a fixed number of columns per row, and that number would have to be as large as the largest string. Therefore, considerable memory could be wasted when storing many strings that are shorter than the longest string. We use string arrays to represent a deck of cards in the next section.

Self Check

(*Fill-In*) A common use of an array of pointers is to form an array of strings, referred to simply as a _____.

Answer: string array.

2 (*True/False*) The characters of the strings in this section's suit array are stored directly in the array's elements.

Answer: False. Though the array appears to contain four strings, each element actually contains the address of the corresponding string's first character. The actual letters and terminating null characters are stored elsewhere in memory.

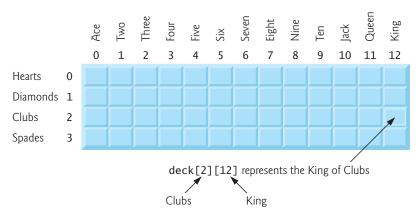
7.11 Random-Number Simulation Case Study: Card Shuffling and Dealing

Let's use random number generation to develop a card shuffling and dealing simulation program, which can then be used to implement programs that play card games. To reveal some subtle performance problems, we've intentionally used suboptimal shuffling and dealing algorithms. In this chapter's exercises and in Chapter 10, we develop more efficient algorithms.

Using the top-down, stepwise refinement approach, we develop a program that will shuffle a deck of 52 playing cards, then deal each card. The top-down approach is particularly useful in attacking more complex problems than you've seen in earlier chapters.

Representing a Deck of Cards as a Two-Dimensional Array

We use a 4-by-13 two-dimensional array deck to represent the deck of playing cards:



The rows correspond to the *suits*—row 0 corresponds to hearts, row 1 to diamonds, row 2 to clubs and row 3 to spades. The columns correspond to the cards' *face* values. Columns 0 through 9 correspond to ace through ten, and columns 10 through 12 correspond to jack, queen and king. We'll load string array suit with character strings representing the four suits, and load string array face with character strings representing the 13 face values.

Shuffling the Two-Dimensional Array

This simulated deck of cards may be *shuffled* as follows. First, set all elements of deck to 0. Then, choose a row (0–3) and a column (0–12) *at random*. Place the number 1 in array element deck[row][column] to indicate that this card will be the first one dealt from the shuffled deck. Repeat this process for the numbers 2, 3, ..., 52, randomly inserting each in the deck array to indicate which cards are to be dealt second, third, ..., and fifty-second in the shuffled deck. As the deck array begins to fill with card numbers, a card may be selected again—i.e., deck[row][column] will be nonzero when it's selected. Ignore this selection and choose other random row and column values repeatedly until you find an *unselected* card. Eventually, the numbers 1 through 52 will occupy the deck array's 52 slots. At that point, the deck of cards is fully shuffled.

Possibility of Indefinite Postponement

This shuffling algorithm can execute *indefinitely* if cards that have already been shuffled are repeatedly selected at random. This phenomenon is known as **indefinite postponement**. In this chapter's exercises, we discuss a better shuffling algorithm that eliminates the possibility of indefinite postponement.

PERF 🔆

Sometimes an algorithm that emerges in a "natural" way can contain subtle performance problems, such as indefinite postponement. Seek algorithms that avoid indefinite postponement.

Dealing Cards from the Two-Dimensional Array

To deal the first card, we search the array for deck[row][column] equal to 1 using nested for statements that vary row from 0 to 3 and column from 0 to 12. What card does that element of the array correspond to? The suit array has been preloaded with the four suits, so to get the card's suit, we print the character string suit[row]. Similarly, to get the card's face, we print the character string face[column]. We also print the character string " of ", as in "King of Clubs", "Ace of Diamonds" and so on.

Developing the Program's Logic with Top-Down, Stepwise Refinement

Let's proceed with the top-down, stepwise refinement process. The *top* is simply:

Shuffle and deal 52 cards

Our first refinement yields:

Initialize the suit array Initialize the face array Initialize the deck array Shuffle the deck Deal 52 cards

"Shuffle the deck" may be refined as follows:

For each of the 52 cards

Place card number in a randomly selected unoccupied element of deck

"Deal 52 cards" may be refined as follows:

For each of the 52 cards

Find the card number in the deck array and print its face and suit

The complete *second refinement* is:

Initialize the suit array

Initialize the face array

Initialize the deck array

For each of the 52 cards

Place card number in a randomly selected unoccupied slot of deck

For each of the 52 cards

Find the card number in the deck array and print the card's face and suit

"Place card number in randomly selected unoccupied slot of deck" may be refined as:

Choose slot of deck randomly

While chosen slot of deck has been previously chosen

Choose slot of deck randomly

Place card number in chosen slot of deck

"Find the card number in the deck array and print its face and suit" may be refined as:

For each slot of the deck array

If slot contains card number

Print the card's face and suit

Incorporating these expansions yields our third refinement:

Initialize the suit array

Initialize the face array

Initialize the deck array

For each of the 52 cards

Choose slot of deck randomly

While slot of deck has been previously chosen

Choose slot of deck randomly

Place card number in chosen slot of deck

For each of the 52 cards

For each slot of deck array

If slot contains desired card number

Print the card's face and suit

This completes the refinement process.

Implementing the Card Shuffling and Dealing Program

The card shuffling and dealing program and a sample execution are shown in Fig. 7.16. When function printf uses the conversion specification %s to print a

string, the corresponding argument must be a pointer to char that points to a string or a char array that contains a string. Line 59's format specification displays the card's face *right-aligned* in a field of five characters followed by " of " and the card's suit *left-aligned* in a field of eight characters. The *minus sign* in %-8s indicates left-alignment.

```
| // fig07_16.c
 2 // Card shuffling and dealing.
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5 #include <time.h>
 7 #define SUITS 4
 8 #define FACES 13
 9 #define CARDS 52
10
// prototypes
12 void shuffle(int deck[][FACES]);
void deal(int deck[][FACES], const char *face[], const char *suit[]);
14
15 int main(void) {
16
       // initialize deck array
17
       int deck[SUITS][FACES] = {0};
18
19
       srand(time(NULL)); // seed random-number generator
20
       shuffle(deck); // shuffle the deck
21
22
       // initialize suit array
23
       const char *suit[SUITS] = {"Hearts", "Diamonds", "Clubs", "Spades"};
24
25
       // initialize face array
       const char *face[FACES] = {"Ace", "Deuce", "Three", "Four", "Five".
26
27
           "Six", "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King"};
28
29
       deal(deck, face, suit); // deal the deck
30
   }
31
32 // shuffle cards in deck
void shuffle(int deck[][FACES]) {
34
       // for each of the cards, choose slot of deck randomly
35
       for (size_t card = 1; card <= CARDS; ++card) {</pre>
36
          size_t row = 0; // row number
37
          size_t column = 0; // column number
38
39
          // choose new random location until unoccupied slot found
40
          do {
41
             row = rand() % SUITS;
42
             column = rand() % FACES;
43
          } while(deck[row][column] != 0);
44
45
          deck[row][column] = card; // place card number in chosen slot
46
       }
47 }
```

Fig. 7.16 | Card shuffling and dealing. (Part 1 of 2.)

```
48
49
    // deal cards in deck
   void deal(int deck[][FACES], const char *face[], const char *suit[]) {
50
51
       // deal each of the cards
52
       for (size_t card = 1; card <= CARDS; ++card) {</pre>
53
          // loop through rows of deck
          for (size_t row = 0; row < SUITS; ++row) {</pre>
54
55
              // loop through columns of deck for current row
              for (size_t column = 0; column < FACES; ++column) {</pre>
56
                 // if slot contains current card, display card
57
                 if (deck[row][column] == card) {
58
                    printf("%5s of %-8s %c", face[column], suit[row],
59
                       card \% 4 == 0 ? : ); // 2-column format
60
61
                 }
62
             }
63
          }
64
       }
65
    }
```

```
Ace of Hearts
                   Jack of Hearts
                                     Five of Clubs
                                                        King of Clubs
Eight of Diamonds Three of Clubs
                                     Deuce of Hearts
                                                        Four of Hearts
 Ace of Clubs
                  Deuce of Spades
                                    Queen of Diamonds
                                                         Six of Hearts
Seven of Clubs
                  Five of Hearts
                                    Deuce of Clubs
                                                        King of Hearts
 Nine of Spades
                  Ace of Spades
                                      Ace of Diamonds Eight of Spades
Eight of Hearts
                   Ten of Spades
                                      Ten of Hearts
                                                       Oueen of Clubs
 Jack of Spades
                   Jack of Diamonds Three of Spades
                                                        Four of Clubs
 Four of Spades
                    Ten of Clubs
                                     King of Diamonds
                                                        Six of Spades
 Nine of Clubs
                    Six of Diamonds Queen of Spades
                                                        King of Spades
 Four of Diamonds Eight of Clubs
                                      Jack of Clubs
                                                       Seven of Hearts
Seven of Diamonds Three of Hearts
                                     Five of Spades
                                                       Nine of Hearts
 Nine of Diamonds Three of Diamonds Deuce of Diamonds Queen of Hearts
  Six of Clubs
                  Seven of Spades
                                     Five of Diamonds
                                                         Ten of Diamonds
```

Fig. 7.16 | Card shuffling and dealing. (Part 2 of 2.)

Improving the Dealing Algorithm

There's a weakness in the dealing algorithm. Once a match is found, the two inner for statements continue searching deck's remaining elements. We correct this deficiency in this chapter's exercises and in a Chapter 10 case study.

Related Exercises

This Card Shuffling and Dealing case study is supported by the following exercises:

- Exercise 7.12 (Card Shuffling and Dealing: Dealing Poker Hands)
- Exercise 7.13 (Project: Card Shuffling and Dealing—Which Poker Hand is Better?)
- Exercise 7.14 (Project: Card Shuffling and Dealing—Simulating the Dealer)
- Exercise 7.15 (Project: Card Shuffling and Dealing—Allowing Players to Draw Cards)

344 Chapter 7 Pointers

 Exercise 7.16 (Card Shuffling and Dealing Modification: High-Performance Shuffle)

Self Check

I (*Fill-In*) The shuffling algorithm we presented can execute indefinitely if cards that have already been shuffled are repeatedly selected at random. This phenomenon is known as _____.

Answer: indefinite postponement.

2 (*True/False*) The format specification "%5s of%-8s" prints a string left-aligned in a field of five characters followed by " of " and a string right-aligned in a field of eight characters.

Answer: False. Actually, this format specification prints a string *right-aligned* in a field of five characters followed by " of " and a string *left-aligned* in a field of eight characters.

7.12 Function Pointers

In Chapter 6, we saw that an array name is really the address in memory of the array's first element. Similarly, a function's name is really the starting address in memory of the code that performs the function's task. A **pointer to a function** contains the *address* of the function in memory. Pointers to functions can be passed to functions, returned from functions, stored in arrays, assigned to other function pointers of the same type and compared with one another for equality or inequality.

7.12.1 Sorting in Ascending or Descending Order

To demonstrate pointers to functions, Fig. 7.17 presents a modified version of Fig. 7.11's bubble-sort program. The new version consists of main and functions bubbleSort, swap, ascending and descending. Function bubbleSort receives a pointer to a function as an argument—either function ascending or function descending—in addition to an int array and the array's size. The user chooses whether to sort the array in ascending (1) or descending (2) order. If the user enters 1, main passes a pointer to function ascending to function bubbleSort. If the user enters 2, main passes a pointer to function descending to function bubbleSort.

```
// fig07_17.c
// Multipurpose sorting program using function pointers.
#include <stdio.h>
#define SIZE 10

// prototypes
void bubbleSort(int work[], size_t size, int (*compare)(int a, int b));
int ascending(int a, int b);
int descending(int a, int b);
```

```
11
    int main(void) {
12
       // initialize unordered array a
13
       int a[SIZE] = \{ 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 \};
14
15
       printf("%s", "Enter 1 to sort in ascending order,\n"
16
                "Enter 2 to sort in descending order: ");
17
       int order = 0;
       scanf("%d", &order);
18
19
20
       puts("\nData items in original order");
21
22
       // output original array
23
       for (size_t counter = 0; counter < SIZE; ++counter) {</pre>
24
          printf("%5d", a[counter]);
25
       }
26
27
       // sort array in ascending order; pass function ascending as an
28
       // argument to specify ascending sorting order
29
       if (order == 1) {
30
          bubbleSort(a, SIZE, ascending);
31
          puts("\nData items in ascending order");
32
33
       else { // pass function descending
          bubbleSort(a, SIZE, descending);
34
35
          puts("\nData items in descending order");
36
       }
37
38
       // output sorted array
39
       for (size_t counter = 0; counter < SIZE; ++counter) {</pre>
40
          printf("%5d", a[counter]);
       }
41
42
43
       puts("\n");
    }
44
45
    // multipurpose bubble sort; parameter compare is a pointer to
46
47
    // the comparison function that determines sorting order
48
    void bubbleSort(int work[], size_t size, int (*compare)(int a, int b)) {
       void swap(int *element1Ptr, int *element2ptr); // prototype
49
50
51
       // loop to control passes
52
       for (int pass = 1; pass < size; ++pass) {</pre>
53
          // loop to control number of comparisons per pass
54
          for (size_t count = 0; count < size - 1; ++count) {</pre>
55
             // if adjacent elements are out of order, swap them
56
             if ((*compare)(work[count], work[count + 1])) {
57
                 swap(&work[count], &work[count + 1]);
             }
58
59
          }
60
       }
    }
61
62
```

Fig. 7.17 | Multipurpose sorting program using function pointers. (Part 2 of 3.)

```
// swap values at memory locations to which element1Ptr and
   // element2Ptr point
65 void swap(int *element1Ptr, int *element2Ptr) {
66
       int hold = *element1Ptr;
       *element1Ptr = *element2Ptr;
67
       *element2Ptr = hold;
68
69
   }
70
71 // determine whether elements are out of order for an ascending order sort
    int ascending(int a, int b) {
72
       return b < a; // should swap if b is less than a
73
74
    }
75
    // determine whether elements are out of order for a descending order sort
77 int descending(int a, int b) {
       return b > a; // should swap if b is greater than a
79
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1
Data items in original order
                                 89
                                          45
                                                37
         6
                   8 10 12
                                      68
              4
Data items in ascending order
         4
              6
                  8
                      10
                                 37
                                     45
                                          68
                                                89
Enter 1 to sort in ascending order,
```

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
2 6 4 8 10 12 89 68 45 37

Data items in descending order
89 68 45 37 12 10 8 6 4 2

Fig. 7.17 Multipurpose sorting program using function pointers. (Part 3 of 3.)

Function Pointer Parameter

The following parameter appears in the function header for bubbleSort (line 48):

```
int (*compare)(int a, int b)
```

This tells bubbleSort to expect a parameter (compare) that's a pointer to a function, specifically for a function that receives two ints and returns an int result. The parentheses around *compare are required to group the * with compare and indicate that compare is a *pointer*. Without the parentheses, the declaration would have been

```
int *compare(int a, int b)
```

which declares a function that receives two integers as parameters and returns a pointer to an integer.

To call the function passed to bubbleSort via its function pointer, we deference it, as shown in the if statement at line 56:

```
if ((*compare)(work[count], work[count + 1]))
```

The call to the function could have been made without dereferencing the pointer as in

```
if (compare(work[count], work[count + 1]))
```

which uses the pointer directly as the function name. The first method of calling a function through a pointer explicitly shows that compare is a pointer to a function that's dereferenced to call the function. The second technique makes it appear that compare is an *actual* function name. This may confuse someone reading the code who'd like to see compare's function definition and finds that it's never defined.

7.12.2 Using Function Pointers to Create a Menu-Driven System

A common use of function pointers is in *menu-driven systems*. A program prompts a user to select an option from a menu (possibly from 0 to 2) by typing the menu item's number. The program services each option with a different function. It stores pointers to each function in an array of function pointers. The user's choice is used as an array subscript, and the pointer in the array is used to call the function.

Figure 7.18 provides a generic example of the mechanics of defining and using an array of function pointers. We define three functions—function1, function2 and function3. Each takes an integer argument and returns nothing. We store pointers to these functions in array f (line 13). Beginning at the leftmost set of parentheses, the definition is read, "f is an array of 3 pointers to functions that each take an int as an argument and return void." The array is initialized with the names of the three functions. When the user enters a value between 0 and 2, we use the value as the subscript into the array of pointers to functions. In the function call (line 23), f[choice] selects the pointer at location choice in the array. We dereference the pointer to call the function, passing choice as the function's argument. Each function prints its argument's value and its function name to show that the function was called correctly. In this chapter's exercises, you'll develop several menu-driven systems.

```
// fig07_18.c
2 // Demonstrating an array of pointers to functions.
3
   #include <stdio.h>
4
5 // prototypes
   void function1(int a);
7
   void function2(int b);
   void function3(int c);
8
9
   int main(void) {
10
       // initialize array of 3 pointers to functions that each take an
11
       // int argument and return void
12
       void (*f[3])(int) = {function1, function2, function3};
13
```

Fig. 7.18 Demonstrating an array of pointers to functions. (Part 1 of 2.)

```
14
       printf("%s", "Enter a number between 0 and 2, 3 to end: ");
15
16
       int choice = 0;
       scanf("%d", &choice);
17
18
19
       // process user
       while (choice >= 0 && choice < 3) {
20
          // invoke function at location choice in array f and pass
21
22
          // choice as an argument
23
          (*f[choice])(choice);
24
          printf("%s", "Enter a number between 0 and 2, 3 to end: ");
25
26
          scanf("%d", &choice);
       }
27
28
29
       puts("Program execution completed.");
30
   }
31
    void function1(int a) {
32
       printf("You entered %d so function1 was called\n\n", a);
33
34
   }
35
   void function2(int b) {
36
       printf("You entered %d so function2 was called\n\n", b);
37
38
    }
39
40 void function3(int c) {
       printf("You entered %d so function3 was called\n\n", c);
41
42 }
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called
Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called
Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called
Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

Fig. 7.18 Demonstrating an array of pointers to functions. (Part 2 of 2.)

Self Check

1 (*True/False*) Consider the following parameter which appeared in the function header for our bubble sort function:

```
int (*compare)(int a, int b)
```

This compare parameter is a pointer to a function that receives two integer parameters and returns an integer result. The parentheses around *compare are optional but we include them for clarity.

Answer: *False*. The parentheses are required to group *compare to indicate that compare is a pointer. Without the parentheses, the declaration would have been

```
int *compare(int a, int b)
```

which simply is the header of a function that receives two integers as parameters and returns a pointer to an integer.

2 (*Fill-In*) Just as a pointer to a variable is dereferenced to access the variable's value, a pointer to a function is dereferenced to _____. **Answer:** call the function.

7.13 Secure C Programming

printf_s, scanf_s and Other Secure Functions

Earlier Secure C Programming sections presented printf_s and scanf_s and mentioned other more secure versions of standard library functions described by Annex K of the C standard. A key feature of functions like printf_s and scanf_s that makes them more secure is that they have *runtime constraints* requiring their pointer arguments to be non-NULL. The functions check these runtime constraints *before* attempting to use the pointers. Any NULL pointer argument is a *constraint violation* and causes the function to fail and return a status notification. A call to scanf_s returns EOF if any of its pointer arguments (including the format-control string) are NULL. A call to printf_s stops outputting data and returns a negative number if the format-control string is NULL or any argument that corresponds to a %s is NULL. For complete details of the Annex K functions, see the C standard document or your compiler's library documentation.

Other CERT Guidelines Regarding Pointers

Misused pointers are the source of many common security vulnerabilities in systems today. CERT provides various guidelines to help you prevent such problems. If you're building industrial-strength C systems, you should familiarize yourself with the CERT C Secure Coding Standard at https://wiki.sei.cmu.edu/. The following guidelines apply to pointer programming techniques that we presented in this chapter:

- EXP34-C: Dereferencing NULL pointers typically causes programs to crash, but CERT has encountered cases in which dereferencing NULL pointers can allow attackers to execute code.
- DCL13-C: Section 7.5 discussed uses of const with pointers. If a function parameter points to a value that will not be changed by the function, const should be used to indicate that the data is constant. For example, to represent a pointer to a string that will not be modified, use const char * as the pointer parameter's type.
- WIN04-C: This guideline discusses techniques for encrypting function pointers on Microsoft Windows to help prevent attackers from overwriting them and executing attack code.

Self Check

(*Fill-In*) A key feature of functions like printf_s and scanf_s is that they have runtime constraints requiring their pointer arguments to be _____.

Answer: non-NULL.

2 (*True/False*) Misused pointers lead to many of the most common security vulnerabilities in systems today.

Answer: True.

Summary

Section 7.2 Pointer Variable Definitions and Initialization

- A **pointer** (p. 310) contains an address of another variable that contains a value. In this sense, a variable name *directly* references a value, and a pointer *indirectly* references a value.
- Referencing a value through a pointer is called **indirection** (p. 311).
- Pointers can be defined to point to objects of any type.
- Pointers should be initialized either when they're defined or in an assignment statement. A pointer may be initialized to NULL, 0 or an address. A pointer with the value NULL points to nothing. Initializing a pointer to 0 is equivalent to initializing a pointer to NULL, but NULL is preferred for clarity. The value 0 is the only integer value that can be assigned directly to a pointer variable.
- NULL is a symbolic constant defined in the <stddef. h> header (and several other headers).

Section 7.3 Pointer Operators

- The &, or address operator (p. 312), is a unary operator that returns its operand's address.
- The operand of the address operator must be a variable.
- The **indirection operator** * (p. 313) returns the value of the object to which its operand points.
- The printf conversion specification %p outputs a memory location as a hexadecimal integer on most platforms.

Section 7.4 Passing Arguments to Functions by Reference

- In C, arguments (other than arrays) are passed by value (p. 315).
- C programs accomplish **pass-by-reference** (p. 315) by using pointers and the indirection operator. To pass a variable by reference, apply the address operator (&) to the variable's name.
- When the address of a variable is passed to a function, the indirection operator (*) may be used in the function to read and/or modify the value at that location in the caller's memory.
- A function receiving an address as an argument must define a **pointer parameter** to receive the address.
- The compiler does not differentiate between a function that receives a pointer and one that receives a **one-dimensional array**. A function must "know" when it's receiving an array vs. a single variable passed by reference.
- When the compiler encounters a function parameter for a one-dimensional array of the form int b[], the compiler converts the parameter to the pointer notation int *b.

Section 7.5 Using the const Qualifier with Pointers

- The **const qualifier** (p. 319) indicates that a variable's value should not be modified.
- There are four ways to pass a pointer to a function (p. 319): a non-constant pointer to non-constant data, a constant pointer to non-constant data, a non-constant pointer to constant data, and a constant pointer to constant data.
- With a non-constant pointer to non-constant data, the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data items.
- A non-constant pointer to constant data can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified.
- A constant pointer to non-constant data always points to the same memory location, and the data at that location can be modified through the pointer. This is the default for an array name.
- A constant pointer to constant data always points to the same memory location, and the data at that memory location cannot be modified.

Section 7.7 sizeof Operator

- Unary operator sizeof (p. 328) determines the size in bytes of a variable or type.
- When applied to an array's name, sizeof returns the array's total number of bytes.
- Operator sizeof can be applied to any variable name, type or value.
- The parentheses used with sizeof are required if a type name is supplied as its operand.

Section 7.8 Pointer Expressions and Pointer Arithmetic

- A limited set of arithmetic operations (p. 331) may be performed on pointers. You can increment (++) or decrement (--) a pointer, add an integer to a pointer (+ or +=), subtract an integer from a pointer (- or -=) and subtract one pointer from another.
- When you add an integer to or subtract an integer from a pointer, the pointer is incremented or decremented by that integer times the size of the object to which the pointer refers.
- Two pointers to elements of the same array may be subtracted from one another to determine the number of elements between them.
- A pointer can be assigned to another pointer if both have the same type. An exception is a void * pointer (p. 332), which can represent any pointer type. All pointer types can be assigned a void * pointer, and a void * pointer can be assigned a pointer of any type.
- A void * pointer cannot be dereferenced.
- Pointers can be compared using equality and relational operators, but such comparisons are meaningless unless the pointers point to elements of the same array. Pointer comparisons compare the addresses stored in the pointers.
- A common use of pointer comparison is determining whether a pointer is NULL.

Section 7.9 Relationship between Pointers and Arrays

- Arrays and pointers are intimately related in C and often may be used interchangeably.
- An array name can be thought of as a constant pointer.
- Pointers can be used to do any operation involving array subscripting.
- When a pointer points to the beginning of an array, adding an **offset** (p. 333) to the pointer indicates which element of the array should be referenced. The offset value is identical to the array subscript. This is referred to as pointer/offset notation.

- An array name can be treated as a pointer and used in pointer arithmetic expressions that do not attempt to modify the pointer's value.
- Pointers can be subscripted (p. 334) like arrays. This is referred to as pointer/subscript notation.
- A parameter of type const char * typically represents a constant string.

Section 7.10 Arrays of Pointers

• Arrays may contain pointers (p. 338). A common use of an array of pointers is to form an array of strings (p. 338). Each element is a string, but a C string is essentially a pointer to its first character. So, each element is actually a pointer to the first character of a string.

Section 7.12 Function Pointers

- A function pointer (p. 347) contains the address of a function in memory. A function name is really the starting address in memory of the code that performs the function's task.
- Pointers to functions can be passed to functions, returned from functions, stored in arrays assigned to other function pointers and compared with one another for equality or inequality.
- A pointer to a function is dereferenced to call the function. A function pointer can be used directly as the function name when calling the function.
- A common use of function pointers is in menu-driven systems.

Self-Review Exercises

- 7.1 Answer each of the following:
 a) A pointer variable contains as its value another variable's _____.
 b) Three values can be used to initialize a pointer ______, ____ and ____.
 c) The only integer that can be assigned to a pointer is ______.
- **7.2** State whether the following are *true* or *false*. If the answer is *false*, explain why.
 - a) A pointer that's declared to be void can be dereferenced.
 - b) Pointers of different types may not be assigned to one another without a cast operation.
- **7.3** Answer each of the following. Assume that single-precision floating-point numbers are stored in four bytes, and that the array's starting address is location 1002500 in memory. Each part of the exercise should use the results of previous parts where appropriate.
 - a) Define a float array called numbers with 10 elements, and initialize the elements to the values 0.0, 1.1, 2.2, ..., 9.9. Assume the symbolic constant SIZE has been defined as 10.
 - b) Define a pointer, nPtr, that points to a float.
 - c) Use a for statement and array subscript notation to print array numbers' elements. Use one digit of precision to the right of the decimal point.
 - d) Give two separate statements that assign the starting address of array numbers to the pointer variable nPtr.

- e) Print numbers' elements using pointer/offset notation with the pointer nPtr.
- f) Print numbers' elements using pointer/offset notation with the array name as the pointer.
- g) Print numbers' elements by subscripting pointer nPtr.
- h) Refer to element 4 of numbers using array subscript notation, pointer/offset notation with the array name as the pointer, pointer subscript notation with nPtr and pointer/offset notation with nPtr.
- i) Assuming that nPtr points to the beginning of array numbers, what address is referenced by nPtr + 8? What value is stored at that location?
- j) Assuming that nPtr points to numbers[5], what address is referenced by nPtr -= 4? What's the value stored at that location?
- 7.4 For each of the following, write a statement that performs the specified task. Assume that float variables number1 and number2 are defined and that number1 is initialized to 7.3.
 - a) Define the variable fPtr to be a pointer to an object of type float.
 - b) Assign the address of variable number1 to pointer variable fPtr.
 - c) Print the value of the object pointed to by fPtr.
 - d) Assign the value of the object pointed to by fPtr to variable number2.
 - e) Print the value of number 2.
 - f) Print the address of number1. Use the %p conversion specification.
 - g) Print the address stored in fPtr. Use the %p conversion specifier. Is the value printed the same as the address of number1?
- **7.5** Do each of the following:
 - a) Write the function header for a function exchange that takes two pointers to floating-point numbers x and y as parameters and does not return a value.
 - b) Write the function prototype for the function in part (a).
 - c) Write the function header for a function evaluate that returns an integer and that takes as parameters integer x and a pointer to function poly, which represents a function that takes an integer parameter and returns an integer.
 - d) Write the function prototype for the function in part (c).
- **7.6** Find the error in each of the following program segments. Assume:

```
int *zPtr; // zPtr will reference array z
int *aPtr = NULL;
void *sPtr = NULL;
int number;
int z[5] = {1, 2, 3, 4, 5};
sPtr = z;
a) ++zptr;
b) // use pointer to get array
number = zPtr;
c) // assign array element 2 to number; assume zPtr is initialized
number = *zPtr[2];
```

```
d) // print entire array z; assume zPtr is initialized
  for (size_t i = 0; i <= 5; ++i) {
     printf("%d ", zPtr[i]);
   }
e) // assign the value pointed to by sPtr to number
   number = *sPtr;
f) ++z;</pre>
```

Answers to Self-Review Exercises

- 7.1 a) address. b) 0, NULL, an address. c) 0.
- **7.2** a) *False*. A pointer to void cannot be dereferenced, because there's no way to know exactly how many bytes of memory to dereference. b) *False*. Pointers of type void can be assigned pointers of other types, and pointers of type void can be assigned to pointers of other types.
- **7.3** See the answers below:

```
a) float numbers[SIZE] =
     \{0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9\};
b) float *nPtr;
c) for (size_t i = 0; i < SIZE; ++i) {
      printf("%.1f ", numbers[i]);
d) nPtr = numbers;
  nPtr = &numbers[0];
e) for (size_t i = 0; i < SIZE; ++i) {
     printf("%.1f ", *(nPtr + i));
f) for (size_t i = 0; i < SIZE; ++i) {
      printf("%.1f ", *(numbers + i));
g) for (size_t i = 0; i < SIZE; ++i) {
     printf("%.1f ", nPtr[i]);
h) numbers [4]
  *(numbers + 4)
  nPtr[4]
  *(nPtr + 4)
i) The address is 1002500 + 8 * 4 = 1002532. The value is 8.8.
i) The address of numbers [5] is 1002500 + 5 * 4 = 1002520.
  The address of nPtr -= 4 is 1002520 - 4 * 4 = 1002504.
  The value at that location is 1.1.
```

7.4 See the answers below:

```
a) float *fPtr;
```

- b) fPtr = &number1; c) printf("The value of *fPtr is %f\n", *fPtr); d) number2 = *fPtr; e) printf("The value of number2 is %f\n", number2); f) printf("The address of number1 is %p\n", &number1); g) printf("The address stored in fptr is %p\n", fPtr); Yes, the value is the same. b) void exchange(float *x, float *y);
- **7.5** a) void exchange(float *x, float *y) c) int evaluate(int x, int (*poly)(int)) d) int evaluate(int x, int (*poly)(int));
- 7.6 a) Error: zPtr has not been initialized. Correction: Initialize zPtr with zPtr = z; before doing pointer arithmetic.
 - b) Error: The pointer is not dereferenced. Correction: Change the statement to number = *zPtr;
 - c) Error: zPtr[2] is not a pointer and should not be dereferenced. Correction: Change *zPtr[2] to zPtr[2].
 - d) Error: Referring to an array element outside the array bounds with pointer subscripting.
 - Correction: Change the operator <= in the for condition to <.
 - e) Error: Dereferencing a void pointer. Correction: To dereference the pointer, it must first be cast to an integer pointer. Change the statement to number = *((int *) sPtr);
 - f) Error: Trying to modify an array name with pointer arithmetic. Correction: Use a pointer variable instead of the array name to accomplish pointer arithmetic, or subscript the array name to refer to a specific element.

Exercises

- Answer each of the following:
 - a) The _____ operator returns its operand's location in memory.
 - b) The _____ operator returns the value of the object to which its operand points.
 - c) To accomplish pass-by-reference when passing a nonarray variable to a function, it's necessary to pass the _____ of the variable to the function.
- 7.8 State whether the following are true or false. If false, explain why.
 - a) Two pointers that point to different arrays cannot be compared meaningfully.
 - b) Because the name of an array is a pointer to the first element of the array, array names may be manipulated in precisely the same manner as pointers.
- Answer each of the following. Assume that integers are stored in four bytes and that the starting address of the array is at location 1002500 in memory.
 - a) Define a five-element int array values, and initialize the elements to the even integers from 2 to 10. Assume the symbolic constant SIZE is defined as 5.

- b) Define a pointer vPtr that points to an object of type int.
- c) Print the elements of array values using array subscript notation. Use a for statement and assume integer control variable i has been defined.
- d) Give two separate statements that assign the starting address of array values to pointer variable vPtr.
- e) Print the elements of array values using pointer/offset notation.
- f) Print the elements of array values using pointer/offset notation with the array name as the pointer.
- g) Print the elements of array values by subscripting the pointer to the array.
- h) Refer to element 4 of values using array subscript notation, pointer/offset notation via the array name, pointer subscript notation, and pointer/offset notation.
- i) What address is referenced by vPtr + 3? What value is stored at that location?
- j) Assuming vPtr points to values [4], what address is referenced by vPtr -= 4? What value is stored at that location?
- **7.10** For each of the following, write a single statement that performs the indicated task. Assume that long integer variables value1 and value2 have been defined and that value1 has been initialized to 200000.
 - a) Define the variable 1Ptr to be a pointer to an object of type long.
 - b) Assign the address of variable value1 to pointer variable 1Ptr.
 - c) Print the value of the object pointed to by 1Ptr.
 - d) Assign the value of the object pointed to by 1Ptr to variable value2.
 - e) Print the value of value2.
 - f) Print the address of value1.
 - g) Print the address stored in 1Ptr. Is the value the same as the address of value1?

7.11 Do each of the following:

- a) Write the function header for function zero, which takes a long integer array parameter bigIntegers and does not return a value.
- b) Write the function prototype for the function in part (a).
- c) Write the function header for function add1AndSum, which takes an integer array parameter oneTooSmall and returns an integer.
- d) Write the function prototype for the function described in part (c).

Note: Exercises 7.12-7.15 are reasonably challenging. Once you have done these problems, you ought to be able to implement most popular card games easily.

- **7.12** (Card Shuffling and Dealing: Dealing Poker Hands) Modify the program in Fig. 7.16 so that the card-dealing function deals a five-card poker hand. Then write the following additional functions:
 - a) Determine whether the hand contains a pair.
 - b) Determine whether the hand contains two pairs.
 - c) Determine whether the hand contains three of a kind (e.g., three jacks).
 - d) Determine whether the hand contains four of a kind (e.g., four aces).

- e) Determine whether the hand contains a flush (i.e., all five cards of the same suit).
- f) Determine whether the hand contains a straight (i.e., five cards of consecutive face values).
- **7.13** (Project: Card Shuffling and Dealing—Which Poker Hand is Better?) Use the functions developed in Exercise 7.12 to write a program that deals two five-card poker hands, evaluates each, and determines which is the better hand.
- **7.14** (Project: Card Shuffling and Dealing—Simulating the Dealer) Modify the program developed in Exercise 7.13 so that it can simulate the dealer. The dealer's five-card hand is dealt "face down" so the player cannot see it. The program should then evaluate the dealer's hand, and based on the quality of the hand, the dealer should draw one, two or three more cards to replace the corresponding number of unneeded cards in the original hand. The program should then reevaluate the dealer's hand. [Caution: This is a difficult problem!]
- **7.15** (Project: Card Shuffling and Dealing—Allowing Player's to Draw Cards) Modify the program developed in Exercise 7.14 so that it can handle the dealer's hand automatically, but the player is allowed to decide which cards of the player's hand to replace. The program should then evaluate both hands and determine who wins. Now use this new program to play 20 games against the computer. Who wins more games, you or the computer? Have one of your friends play 20 games against the computer. Who wins more games? Based on these games' results, refine your poker-playing program (this, too, is a difficult problem). Play 20 more games. Does your modified program play a better game?
- **7.16** (Card Shuffling and Dealing Modification: High-Performance Shuffle) In Fig. 7.16, we intentionally used an inefficient card shuffling algorithm with the possibility of indefinite postponement. In this problem, you'll create a high-performance shuffling algorithm that avoids indefinite postponement.

Modify the program of Fig. 7.16 as follows. Begin by initializing the deck array as shown below:

Unshuffled array													
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	14	15	16	17	18	19	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47	48	49	50	51	52

Modify the shuffle function to loop row-by-row and column-by-column through the array, touching every element once. Each element should be swapped with a randomly selected element of the array. Print the resulting array to determine whether the deck is satisfactorily shuffled. The following is a sample set of shuffled values:

Sample shuffled array													
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2	44
1	13	28	14	16	21	30	8	11	31	17	24	7	1
2	12	33	15	42	43	23	45	3	29	32	4	47	26
3	50	38	52	39	48	51	9	5	37	49	22	6	20

You may want your program to call the shuffle function several times to ensure a satisfactory shuffle.

Although the approach in this problem improves the shuffling algorithm, the dealing algorithm still requires searching the deck array for card 1, then card 2, then card 3, and so on. Worse yet, even after the dealing algorithm locates and deals the card, the algorithm still searches through the remainder of the deck. Modify the program of Fig. 7.16 so that once a card is dealt, no further attempts are made to match that card number, and the program immediately proceeds with dealing the next card. In Chapter 10, we develop a dealing algorithm that requires only one operation per card.

7.17 What does this program do, assuming the user enters two strings of the same length?

```
I // ex07_19.c
 2 // What does this program do?
 3 #include <stdio.h>
 4 #define SIZE 80
 5
 6
   void mystery1(char *s1, const char *s2); // prototype
 7
 8
    int main(void) {
 9
       char string1[SIZE]; // create char array
10
       char string2[SIZE]; // create char array
11
12
       puts("Enter two strings: ");
13
       scanf("%39s%39s" , string1, string2);
14
       mystery1(string1, string2);
15
       printf("%s", string1);
16
    }
17
18
    // What does this function do?
19
    void mystery1(char *s1, const char *s2) {
20
       while (*s1 != ) {
21
          ++s1;
22
       }
23
       for (; *s1 = *s2; ++s1, ++s2) {
24
25
          ; // empty statement
26
27
    }
```

7.18 What does this program do?

```
I // ex07_20.c
   // what does this program do?
 3
   #include <stdio.h>
 4 #define SIZE 80
 5
 6
    size_t mystery2(const char *s); // prototype
 7
    int main(void) {
 8
 9
       char string[SIZE]; // create char array
10
П
       puts("Enter a string: ");
       scanf("%79s", string);
12
13
       printf("%d\n", mystery2(string));
14
15
   // What does this function do?
16
17
    size_t mystery2(const char *s) {
18
       size_t x;
19
       // loop through string
20
       for (x = 0; *s != ; ++s) {
21
22
          ++X;
23
       }
24
25
       return x;
26 }
```

7.19 Find the error in each of the following program segments. If the error can be corrected, explain how.

```
a) int *number;
  printf("%d\n", *number);
b) float *realPtr;
  long *integerPtr;
  integerPtr = realPtr;
c) int * x, y;
  x = y;
d) char s[] = "this is a character array";
  int count;
  for (; *s != ; ++s) {
     printf("%c ", *s);
  }
e) short *numPtr, result;
  void *genericPtr = numPtr;
  result = *genericPtr + 7;
f) float x = 19.34;
  float xPtr = &x;
  printf("%f\n", xPtr);
```

```
g) char *s;
    printf("%s\n", s);
```

7.20 (*Maze Traversal*) The following grid is a two-dimensional array representation of a maze. The # symbols represent the maze's walls, and the periods (.) represent squares in the possible paths through the maze.

The Wikipedia page https://en.wikipedia.org/wiki/Maze_solving_algorithm lists several algorithms for finding a maze's exit. A simple algorithm for walking through a maze guarantees finding the exit (assuming there's an exit). Place your right hand on the wall to your right, and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand from the wall, eventually you'll arrive at the maze's exit. If there's not an exit, you'll eventually arrive back at the starting location. There may be a shorter path than the one you've taken, but you're guaranteed to get out of the maze.

Write recursive function mazeTraverse to walk through the maze. The function should receive as arguments a 12-by-12 character array representing the maze and the maze's starting location. As mazeTraverse attempts to locate the exit from the maze, it should place the character X in each square in the path. The function should display the maze after each move so the user can watch as the maze is solved.

- **7.21** (Generating Mazes Randomly) Write a function mazeGenerator that takes as an argument a two-dimensional 12-by-12 character array and randomly produces a maze. The function should also provide the starting and ending locations of the maze. Try your function mazeTraverse from Exercise 7.20 using several randomly generated mazes.
- **7.22** (*Mazes of Any Size*) Generalize functions mazeTraverse and mazeGenerator of Exercises 7.20–7.21 to process mazes of any width and height.
- **7.23** What does this program do, assuming that the user enters two strings of the same length?

```
// ex07_26.c
// What does this program do?
```

^{3 #}include <stdio.h>

```
4
    #define SIZE 80
 5
 6
    int mystery3(const char *s1, const char *s2); // prototype
 7
 8
    int main(void) {
       char string1[SIZE]; // create char array
 9
       char string2[SIZE]; // create char array
10
П
12
       puts("Enter two strings: ");
       scanf("%79s%79s", string1 , string2);
13
       printf("The result is %d\n", mystery3(string1, string2));
14
    }
15
16
17
    int mystery3(const char *s1, const char *s2) {
18
       int result = 1;
19
20
       for (; *s1 != && *s2 != ; ++s1, ++s2) {
21
          if (*s1 != *s2) {
22
             result = 0;
23
24
       }
25
26
       return result;
27 }
```

Arrays of Function Pointers

7.24 (Arrays of Function Pointers) Rewrite the program of Fig. 6.17 to use a menudriven interface. The program should offer the user four options as follows:

```
Enter a choice:

O Print the array of grades

1 Find the minimum grade

2 Find the maximum grade

3 Print the average on all tests for each student

4 End program
```

One restriction on using arrays of pointers to functions is that all the pointers must have the same type. The pointers must be to functions of the same return type that receive arguments of the same type. For this reason, the functions in Fig. 6.17 must be modified so that they each return the same type and take the same parameters. Modify functions minimum and maximum to print the minimum or maximum value and return nothing. For option 3, modify function average of Fig. 6.17 to output the average for each student (not a specific student). Function average should return nothing and take the same parameters as printArray, minimum and maximum. Store the pointers to the four functions in array processGrades and use the choice made by the user as the subscript into the array for calling each function.

7.25 (Calculating Circle Circumference, Circle Area or Sphere Volume Using Function Pointers) Using the techniques from Fig. 7.18, create a menu-driven program.

Allow the user to choose whether to calculate a circle's circumference, a circle's area or a sphere's volume. The program should then input a radius from the user, perform the appropriate calculation and display the result. Use an array of function pointers in which each pointer represents a function that returns void and receives a double parameter. The corresponding functions should each display messages indicating which calculation was performed, the value of the radius and the result of the calculation.

7.26 (Calculator Using Function Pointers) Using the techniques you learned in Fig. 7.18, create a menu-driven program that allows the user to choose whether to add, subtract, multiply or divide two numbers. The program should then input two double values from the user, perform the appropriate calculation and display the result. Use an array of function pointers in which each pointer represents a function that returns void and receives two double parameters. The corresponding functions should each display messages indicating which calculation was performed, the values of the parameters and the result of the calculation.

7.27 (Carbon Footprint Calculator) Using arrays of function pointers, as you learned in this chapter, you can specify a set of functions that are called with the same types of arguments and return the same type of data. Governments and companies worldwide are becoming increasingly concerned with carbon footprints (annual releases of carbon dioxide into the atmosphere) from buildings burning various types of fuels for heat, vehicles burning fuels for power, and the like. Many scientists blame these greenhouse gases for the phenomenon called global warming. Create three functions that help calculate the carbon footprint of a building, a car and a bicycle, respectively. Each function should input appropriate data from the user, then calculate and display the carbon footprint. (Check out a few websites that explain how to calculate carbon footprints.) Each function should receive no parameters and return void. Write a program that prompts the user to enter the type of carbon footprint to calculate, then calls the corresponding function in the array of function pointers. For each type of carbon footprint, display some identifying information and the object's carbon footprint.

Special Section—Building Your Own Computer as a Virtual Machine

In the next several exercises, we take a temporary diversion away from the world of high-level language programming. We "peel open" a fake simple computer and look at its internal structure. We introduce machine-language programming for this computer and write several machine-language programs. To make this an especially valuable experience, we then build a software-based *simulation* of this computer on which you actually can execute your machine-language programs! Such a simulated computer is often called a virtual machine.

7.28 (*Machine-Language Programming*) Let's create a computer we'll call the Simpletron. As its name implies, it's a simple machine, but as we'll soon see, it's a powerful one as well. The Simpletron runs programs written in the only language it directly understands—that is, **Simpletron Machine Language**, or **SML** for short.

The Simpletron contains an accumulator—a "special register" in which information is put before the Simpletron uses that information in calculations or examines it in various ways. All information in the Simpletron is handled in terms of words. A word is a signed four-digit decimal number such as +3364, -1293, +0007, -0001, etc. The Simpletron is equipped with a 100-word memory, and these words are referenced by their location numbers 00, 01, ..., 99.

Before running an SML program, we must load or place the program into memory. The first instruction (or statement) of every SML program is always placed in location 00.

Each SML instruction occupies one word of the Simpletron's memory, so instructions are signed four-digit decimal numbers. We assume an SML instruction's sign is always plus, but a data word's sign may be plus or minus. Each Simpletron memory location may contain an instruction, a data value used by a program or an unused (and hence undefined) area of memory. Each SML instruction's first two digits are the operation code specifying the operation to perform. The SML operation codes are summarized in the following table:

Operation code	Meaning	
Input/output operations:		
#define READ 10	Read a word from the keyboard into a specific location in memory.	
#define WRITE 11	Write a word from a specific location in memory to the screen.	
Load/store operations:		
#define LOAD 20	Load a word from a specific location in memory into the accumulator.	
#define STORE 21	Store a word from the accumulator into a specific location in memory.	
Arithmetic operations:		
#define ADD 30	Add a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator).	
#define SUBTRACT 31	Subtract a word from a specific location in memory from the word in the accumulator (leave the result in the accumulator).	
#define DIVIDE 32	Divide a word from a specific location in memory into the word in the accumulator (leave the result in the accumulator).	
#define MULTIPLY 33	Multiply a word from a specific location in memory by the word in the accumulator (leave the result in the accumulator).	
Transfer-of-control operations:		
#define BRANCH 40	Branch to a specific location in memory.	
#define BRANCHNEG 41	Branch to a specific location in memory if the accumulator is negative.	
#define BRANCHZERO 42	Branch to a specific location in memory if the accumulator is zero.	
#define HALT 43	Halt—i.e., the program has completed its task.	

An SML instruction's last two digits are the **operand**—the memory location containing the word to which the operation applies.

Sample SML Program That Adds Two Numbers

Let's consider several simple SML programs. The following SML program reads two numbers from the keyboard, then computes and prints their sum:

Location	Number	Instruction
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

The instruction +1007 reads the first number from the keyboard and places it into location 07. Then +1008 reads the next number into location 08. The *load* instruction, +2007, copies the first number into the accumulator. The *add* instruction, +3008, adds the second number to the number in the accumulator. *All SML arithmetic instructions leave their results in the accumulator.* The *store* instruction, +2109, copies the result from the accumulator into memory location 09, from which the *write* instruction, +1109, then takes the number and prints it as a signed four-digit decimal number to the screen. The *halt* instruction, +4300, terminates execution.

Sample SML Program That Determines the Largest of Two Values

The next SML program reads two numbers from the keyboard, then determines and prints the larger value:

Location	Number	Instruction
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Branch negative to 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

The instruction +4107 is a conditional transfer of control, like an if statement.

Now write SML programs to accomplish each of the following tasks.

- a) Use a sentinel-controlled loop to read positive integers, then compute and print their sum.
- b) Use a counter-controlled loop to read seven numbers, some positive and some negative. Compute and print their average.
- c) Read a series of numbers. Determine and print the largest number. The first number read indicates how many numbers should be processed.

7.29 (A Computer Simulator) It may at first seem outrageous, but in this exercise you'll build your own computer. No, you won't be soldering components together. Rather, you'll use the powerful technique of software-based simulation to create a software model of the Simpletron. You will not be disappointed. Your Simpletron simulator will turn the computer you're using into a Simpletron, and you'll actually be able to run, test and debug the SML programs you wrote in Exercise 7.28!

When you run your Simpletron simulator, it should begin by printing:

```
*** Welcome to Simpletron

***

*** Please enter your program one instruction ***

*** (or data word) at a time. I will type the ***

*** location number and a question mark (?).

***

*** You then type the word for that location.

***

*** Type the sentinel -99999 to stop entering ***

*** your program.
```

Simulate the memory of the Simpletron with a 100-element one-dimensional array memory. Now assume that the simulator is running, and let's examine the dialog as we enter the program of Example 2 of Exercise 7.28:

```
00 ? +1009

01 ? +1010

02 ? +2009

03 ? +3110

04 ? +4107

05 ? +1109

06 ? +4300

07 ? +1110

08 ? +4300

09 ? +0000

10 ? +0000

11 ? -99999

*** Program loading completed ***

*** Program execution begins ***
```

The SML program has now been placed (or loaded) into the array memory. Next, the Simpletron executes the SML program. It begins with the instruction in location

00 and continues sequentially, unless directed to some other part of the program by a transfer of control.

Use the variable accumulator to represent the accumulator register. Use the variable instructionCounter to store the number of the memory location (00 to 99) containing the instruction being performed. Use the variable operationCode to store the operation currently being performed (the instruction word's left two digits). Use the variable operand to store the number of the memory location on which the current instruction operates. Thus, if an instruction has an operand, it's the rightmost two digits of the instruction currently being performed. Do not execute instructions directly from memory. Rather, transfer the next instruction to be performed from memory to a variable called instructionRegister. Then "pick off" the left two digits and place them in the variable operationCode, and "pick off" the right two digits and place them in operand.

When Simpletron begins execution, the special registers are initialized as follows:

```
accumulator +0000 instructionCounter 00 instructionRegister +0000 operationCode 00 operand 00
```

Now let's "walk through" the execution of the first SML instruction, +1009 in memory location 00. This process is called an instruction execution cycle.

The instructionCounter tells us the location of the next instruction to be performed. We *fetch* the contents of that location from memory by using the C statement

```
instructionRegister = memory[instructionCounter];
```

The operation code and the operand are extracted from the instruction register by the statements

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Now the Simpletron must determine that the operation code is actually a *read* (versus a *write*, a *load*, and so on). A switch statement differentiates among the twelve operations of SML. The switch simulates the behavior of various SML instructions as follows (we leave the others to the reader):

```
read: scanf("%d", &memory[operand]);
load: accumulator = memory[operand];
add: accumulator += memory[operand];
Various branch instructions: We'll discuss these shortly.
halt: This instruction prints the message
    *** Simpletron execution terminated ***
```

then prints the name and contents of each register as well as the complete contents of all 100 memory locations. Such a printout is often called a **computer dump**. To help you program your dump function, the output below shows a sample dump:

```
REGISTERS:
accumulator
                          +0000
instructionCounter
                             00
instructionRegister
                          +0000
operationCode
                             00
operand
                             00
MEMORY:
        0
               1
                       2
                              3
                                             5
                                                     6
                                                            7
                                                                   8
                                                                           9
                  +0000
                          +0000
 0
   +0000
           +0000
                                 +0000
                                         +0000
                                                +0000
                                                       +0000
                                                               +0000
                                                                      +0000
10
   +0000
           +0000
                   +0000
                          +0000
                                 +0000
                                         +0000
                                                +0000
                                                        +0000
                                                               +0000
                                                                       +0000
20
    +0000
           +0000
                   +0000
                          +0000
                                  +0000
                                         +0000
                                                +0000
                                                        +0000
                                                               +0000
                                                                       +0000
30
   +0000
           +0000
                   +0000
                          +0000
                                 +0000
                                         +0000
                                                +0000
                                                        +0000
                                                               +0000
                                                                       +0000
40
   +0000
           +0000
                   +0000
                          +0000
                                 +0000
                                         +0000
                                                +0000
                                                        +0000
                                                               +0000
                                                                       +0000
50
   +0000
           +0000
                   +0000
                          +0000
                                 +0000
                                         +0000
                                                +0000
                                                        +0000
                                                               +0000
                                                                      +0000
           +0000
                          +0000
60
   +0000
                   +0000
                                 +0000
                                         +0000
                                                +0000
                                                        +0000
                                                               +0000
                                                                       +0000
                          +0000
70
   +0000
           +0000
                   +0000
                                 +0000
                                         +0000
                                                +0000
                                                        +0000
                                                               +0000
                                                                       +0000
80
   +0000
           +0000
                   +0000
                          +0000
                                 +0000
                                         +0000
                                                +0000
                                                        +0000
                                                               +0000
                                                                       +0000
90 +0000
           +0000
                   +0000
                          +0000
                                 +0000
                                         +0000
                                                +0000
                                                        +0000
                                                               +0000
                                                                       +0000
```

A dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated. You can print leading 0s in front of an integer that is shorter than its field width by placing the 0 formatting flag before the field width in the format specifier as in "%02d". You can place a + or - sign before a value with the + formatting flag. So to produce a number of the form +0000, you can use the format specifier "%+05d".

Let's proceed with the execution of our program's first instruction, namely the +1009 in location 00. As we've indicated, the switch statement simulates this by performing the statement

```
scanf("%d", &memory[operand]);
```

A question mark (?) should be displayed on the screen before the scanf is executed to prompt the user for input. The Simpletron waits for the user to type a value and then press the *Return* (or *Enter*) key. The value is then read into location 09.

At this point, simulation of the first instruction is completed. All that remains is to prepare the Simpletron to execute the next instruction. Because the instruction just performed was not a transfer of control, we need merely increment the instruction counter register as follows:

```
++instructionCounter;
```

This completes the simulated execution of the first instruction. The entire process (i.e., the instruction execution cycle) begins anew with the fetch of the next instruction to be executed.

Now let's consider how the branching instructions—the transfers of control—are simulated. All we need to do is adjust the value in the instruction counter appropriately. Therefore, the unconditional branch instruction (40) is simulated within the switch as

```
instructionCounter = operand;
```

The conditional "branch if accumulator is zero" instruction is simulated as

```
if (accumulator == 0) {
  instructionCounter = operand;
}
```

At this point, you should implement your Simpletron simulator and run the SML programs you wrote in Exercise 7.28. You may embellish SML with additional features and provide for these in your simulator. Exercise 7.30 lists several possible embellishments.

Your simulator should check for various types of errors. During the program loading phase, for example, each number the user types into the Simpletron's memory must be in the range -9999 to +9999. Your simulator should use a while loop to test that each number entered is in this range, and, if not, keep prompting the user to reenter the number until a correct number is entered.

During the execution phase, your simulator should check for serious errors, such as attempts to divide by zero, attempts to execute an invalid operation code and accumulator overflows (i.e., arithmetic operations resulting in values larger than +9999 or smaller than -9999). Such serious errors are fatal errors. When a fatal error is detected, print an error message such as:

and print a full computer dump in the format we've discussed previously. This will help the user locate the error in the program.

Implementation Note: When you implement the Simpletron Simulator, define the memory array and all the registers as variables in main. The program should contain three other functions—load, execute and dump. Function load reads the SML instructions from the user at the keyboard. (Once you study file processing in Chapter 11, you'll be able to read the SML instruction from a file.) Function execute executes the SML program currently loaded in the memory array. Function dump displays the contents of memory and all of the registers stored in main's variables. Pass the memory array and registers to the other functions as necessary to complete their tasks. Functions load and execute need to modify variables that are defined in main, so you'll need to pass those variables to the functions by reference using pointers. You'll need to modify the statements we showed throughout this problem description to use the appropriate pointer notations.

- **7.30** (Modifications to the Simpletron Simulator) In this exercise, we propose several modifications and enhancements to Exercise 7.29's Simpletron Simulator. In Exercises 12.24 and 12.25, we propose building a compiler that converts programs written in a high-level programming language (a variation of BASIC) to Simpletron Machine Language. Some of the following modifications and enhancements may be required to execute the programs produced by the compiler:
 - a) Extend the Simpletron Simulator's memory to contain 1000 memory locations (000 to 999) to enable the Simpletron to handle larger programs.

- b) Allow the simulator to perform remainder calculations. This requires an additional Simpletron Machine Language instruction.
- c) Allow the simulator to perform exponentiation calculations. This requires an additional Simpletron Machine Language instruction.
- d) Modify the simulator to use hexadecimal values rather than integer values to represent Simpletron Machine Language instructions. Online Appendix E, Number Systems, discusses hexadecimal.
- e) Modify the simulator to allow output of a newline. This requires an additional Simpletron Machine Language instruction.
- f) Modify the simulator to process floating-point values in addition to integer values.
- g) Modify the simulator to detect division by 0 logic errors.
- h) Modify the simulator to detect arithmetic-overflow errors.
- i) Modify the simulator to handle string input. [Hint: Each Simpletron word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII decimal equivalent of a character. Add a machine-language instruction that will input a string and store it beginning at a specific Simpletron memory location. The first half of the word at that location will be a count of the number of characters in the string (i.e., the length of the string). Each succeeding half word contains one ASCII character expressed as two decimal digits. The machine-language instruction converts each character into its ASCII equivalent and assigns it to either a left or right half word.]
- j) Modify the simulator to handle output of strings stored in the format of part (g). [Hint: Add a machine-language instruction that prints a string beginning at a specified Simpletron memory location. The first half of the word at that location is the length of the string in characters. Each succeeding half word contains one ASCII character expressed as two decimal digits. The machine-language instruction checks the length and prints the string by translating each two-digit number into its equivalent character.]

Special Section—Embedded Systems Programming Case Study: Robotics with the Webots Simulator

7.31 Webots^{5,6} is an open-source, full-color, 3D, robotics simulator with console-game-quality graphics. It enables you to create a virtual reality in which robots interact with simulated real-world environments. It runs on Windows, macOS and Linux. The simulator is widely used in industry and research to test robots' viability and de-

^{5. &}quot;Webots Open Source Robot Simulator." Accessed December 11, 2020. https://cyberbotics.com.

^{6.} The Webots environment screen captures in this case study are Copyright 2020 Cyberbotics Ltd., which is licensed under the Apache License, Version 2.0 (http://www.apache.org/licenses/LICENSE-2.0).

velop controller software for those robots. Webots uses an Apache open-source license. The following is from their license webpage:⁷

"Webots is released under the terms of the Apache 2.0 license agreement. Apache 2.0 is a [sic] industry friendly, non-contaminating, permissive open source license that grants everyone the right to use a software source code, free of charge, for any purpose, including commercial applications."

Webots was first developed in 1996 at the Swiss Federal Institute of Technology (EPFL). In 1998, the EPFL spin-off Cyberbotics Ltd. was founded to take over the Webots simulator development. Until 2018, Webots was sold as proprietary licensed software. In 2018, Cyberbotics open-sourced Webots under the Apache 2.0 license.^{8,9}

Robotics is not typically discussed in introductory programming textbooks, but Webots makes it easy. Webots comes bundled with simulations for dozens of today's most popular real-world robots that walk, fly, roll, drive and more. For a current list of bundled robots, visit

https://cyberbotics.com/doc/guide/robots

Self-Contained Development Environment

Webots is a **self-contained robotics-simulation environment** with everything you need to begin developing and experimenting with robotics. It includes:

- an interactive 3D simulation area for viewing and interacting with simulations,
- a **code editor** where you can view the bundled simulation code, modify it and write your own, and
- **compilers and interpreters** that enable you to write Webots code in C, C++, Java, Python and MatLab.

You can easily modify existing simulations and re-run their code to see the effects of your changes. You also can develop entirely new simulations, as you'll do in this case study.

Installing Webots

First, check the Webots system requirements at

https://cyberbotics.com/doc/guide/system-requirements

You can download the Webots installer for Windows, macOS or Linux from the Cyberbotics home page:

https://cyberbotics.com/

Once downloaded, run the installer and follow the on-screen prompts.

^{7. &}quot;Webots User Guide R2020b revision 2—License Agreement." Accessed December 14, 2020. https://cyberbotics.com/doc/guide/webots-license-agreement.

^{8. &}quot;Cyberbotics." Accessed December 13, 2020. https://cyberbotics.com/#cyberbotics.

^{9. &}quot;Webots." Accessed December 13, 2020. https://en.wikipedia.org/wiki/Webots.

Guided Tour

Once the installation completes, run the Webots application on your system. You can get a quick overview of many bundled robotics simulations and the robots' capabilities in the Webots guided tour. To do so, select Help > Webots Guided Tour... (the first time you open the Webots environment, this guided tour will begin automatically). Then, in the window that appears (Fig. 7.19), check the Auto checkbox and click Next. This will begin the automated guided tour, which will show you each demonstration for a short time, then switch to the next. The Guided Tour - Webots window displays a brief description of each simulation. Figure 7.19 shows the description for the third simulation, which appears in Fig. 7.20.

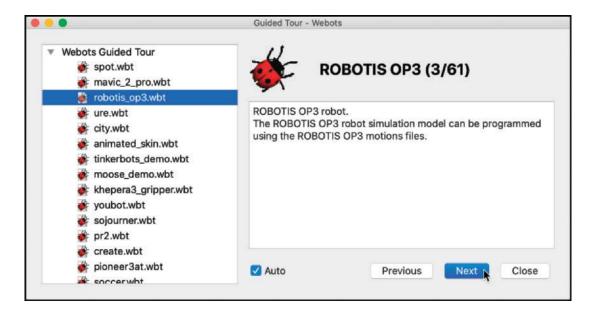


Fig. 7.19 A brief description of the ROBOTIS OP3 robot shown in Fig. 7.20. [The screen captures in this case study are Copyright 2020 Cyberbotics Ltd., which is licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0.]

As the guided tour overviews each simulation, you'll see it live in the Webots environment's 3D viewing area—shown in the center portion of Fig. 7.20. As you'll soon see:

- the environment's left side enables you to manage and configure the components in your robotics simulations, and
- the environment's right side provides an integrated code editor for writing and compiling the C code that controls a robot in your simulation.

We narrowed the code editor for this screen capture. As in most IDEs, the areas within the window can be resized by dragging the divider bars.

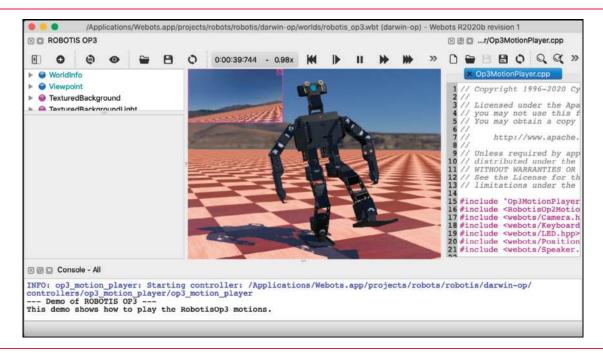


Fig. 7.20 A ROBOTIS OP3 robotics simulation running in Webots.

User Interface Overview

Familiarize yourself with the Webots environment's user interface by reading the following overview:

https://cyberbotics.com/doc/guide/the-user-interface

Webots Tutorial 1: Your First Simulation in Webots

The Webots team provides eight tutorials that introduce you to many aspects of the Webots environment and its robotics-simulation capabilities:

- Tutorial 1: Your First Simulation in Webots
- Tutorial 2: Modification of the Environment
- Tutorial 3: Appearance
- Tutorial 4: More about Controllers
- Tutorial 5: Compound Solid and Physics Attributes
- Tutorial 6: 4-Wheels Robot
- Tutorial 7: Your First PROTO
- Tutorial 8: Using ROS

In this case study, you'll follow their first tutorial to create a robotics simulation using several predefined items:

- a RectangleArena in which your robot will roam,
- several WoodenBox obstacles, and

 an e-puck robot—a simple simulated robot that will move around the RectangleArena and change directions when it encounters a WoodenBox or a wall.

The e-puck simulator corresponds to a real-world, educational robot ¹⁰ that has:

- two independently controlled wheels (known as differential wheels), so the robot can change direction by moving its wheels at different speeds,
- a camera (the upper-left corner of the Webots environment's 3D viewing area shows a small window in which you can view what a robot "sees"),
- eight distance sensors, and
- 10 LED lights with controllable intensity—other e-puck robots can "see" these via their camera for visual interactions between multiple e-puck robots.

This tutorial focuses on using the wheels to make the robot move. You can work through Webots Tutorials 2–4 to use additional e-puck features. For more information on the e-puck robot, visit:

http://www.e-puck.org/

Tutorial Steps

You can find the first Webots tutorial at

https://cyberbotics.com/doc/guide/tutorial-1-your-first-simulationin-webots

Below, we overview each tutorial step, provide additional insights and clarify some of the tutorial instructions.¹¹ The step numbers in the rest of this exercise correspond to the "Hands-on" steps in the Webots tutorial. For each step, you should:

- 1. read the Webots tutorial step,
- 2. read our additional comments for that step, then
- **3.** perform the step's task(s).

The tutorial shows only one diagram of the robotics simulation you'll create. To help you work through the tutorial, we include additional screen captures¹² to clarify the tutorial instructions. As you work through the steps, be sure to save your changes after each modification. If you have to reset the simulation, it will revert to the last saved version.

^{10. &}quot;The e-puck, a Robot Designed for Education in Engineering." Accessed December 13, 2020. https://infoscience.epfl.ch/record/135236?ln=en.

^{11.} This discussion was written in December 2020. The software, documentation and tutorials could change. If you run into problems, visit the Webots forum (https://discord.com/invite/nTWbN9m) and check the Webots StackOverflow questions (https://stackoverflow.com/questions/tagged/webots) or e-mail us at deitel@deitel.com.

^{12.} We changed the background color in our version of the simulation so that the screen captures would be more readable in print.

Step 1: Launch the Webots Application

In **Step 1**, you'll simply open the Webots environment. You'll then perform the additional steps discussed below.

Step 2: Create Your Virtual World

A world defines the simulation environment and is stored in a .wbt file. Internally, this file uses Virtual Reality Modeling Language (VRML) to describe your world's elements. Each world can have characteristics, such as gravity, that affect object interactions. The world specifies the area in which your robot can roam and the objects with which it can interact. The Wizards menu's Create a Webots project directory wizard (Figs. 7.21–7.24) will guide you through setting up a new world with Webots' required folder structure.

In the first screen of the Create a Webots project directory wizard (Fig. 7.21), simply click Continue (or Next) to move to the next step.

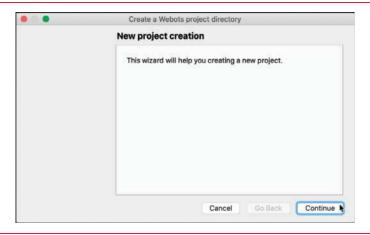


Fig. 7.21 | Initial Create a Webots project directory wizard window.

In the wizard's **Directory selection** step (Fig. 7.22), change your project's directory name from my_project (the default) to my_first_simulation—this folder's default location is your user account's **Documents** folder.¹³

In the wizard's **World settings** step (Fig. 7.23), you'll change the world's filename from empty.wbt (the default) to my_first_simulation.wbt and ensure that all four checkboxes are checked. This will add several Webots predefined components to your virtual world.

The wizard's **Conclusion** step (Fig. 7.24) shows all the folders and files the wizard will generate for your simulation. In subsequent steps, you'll add obstacles and an epuck robot, configure various settings and write some C code that controls the robot.

^{13.} You can change the location where your Webots project is stored. Ours is stored in our user account's /Users/pauldeitel/Documents folder, which you'll see in several screen captures.

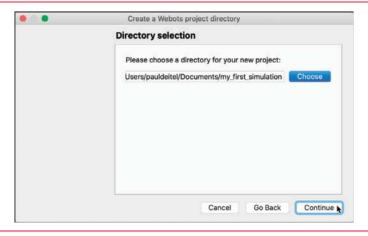


Fig. 7.22 Changing the default project directory name to my_first_simulation.

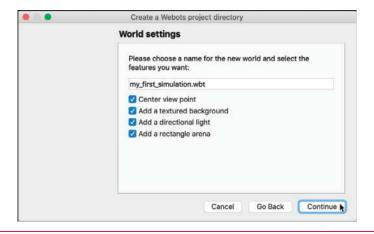


Fig. 7.23 Changing the default world filename and ensuring all checkboxes are checked.

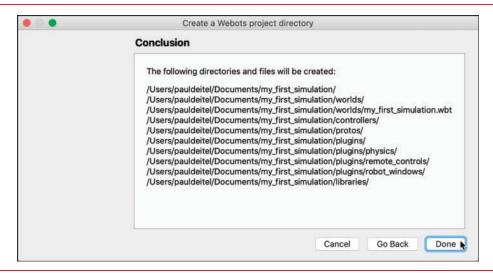


Fig. 7.24 | Summary of the folders and files the wizard will generate for your simulation.

376 Chapter 7 Pointers

When you click Done (macOS) or Finish (Windows and Linux) in the Create a Webots project directory wizard (Fig. 7.24), the Webots 3D viewing area displays an empty RectangleArena with a checkered floor—the default for a RectangleArena (Fig. 7.25). There are several floor-style options. You can experiment with these after you learn how to change settings for the elements in your world. In the 3D viewing area, you can zoom in and out using your mouse wheel, and you can click and drag the RectangleArena to view it from different angles and rotate it.

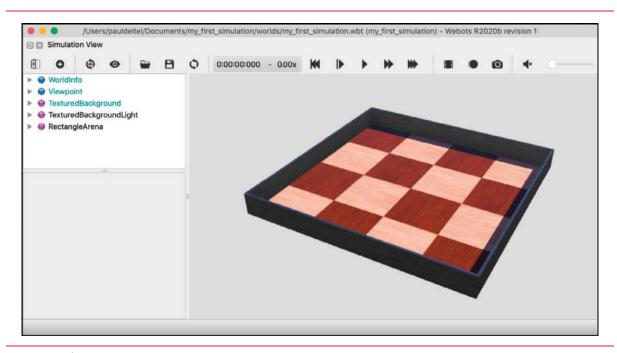


Fig. 7.25 | Initial view of the RectangleArena after completing the Create a Webots project directory wizard.

Step 3: Modify the RectangleArena

Each element in the simulation is a node in the world's scene tree, which you can see at the left side of the Webots environment. In this step, you'll select a node in the scene tree, then change some of its settings, called **fields**. After you perform this step, the environment should appear as in Fig. 7.26.

The colored arrows over the RectangleArena appear for any object you select in the world, either by clicking it in the 3D viewing area or by clicking its node in the scene tree. You can drag these arrowheads to move, rotate and tilt the object. ¹⁴ Dragging a straight arrowhead moves the object along that axis. Dragging a circular arrowhead rotates or tilts the object around that axis. Save your environment, then experiment with these arrowheads to see how they affect the RectangleArena's position. You can then reset the environment to restore the original position.

^{14. &}quot;Webots User Guide R2020b revision 2 — The 3D Window — Moving a Solid Object." Accessed December 13, 2020. https://cyberbotics.com/doc/guide/the-3d-window#moving-a-solid-object.

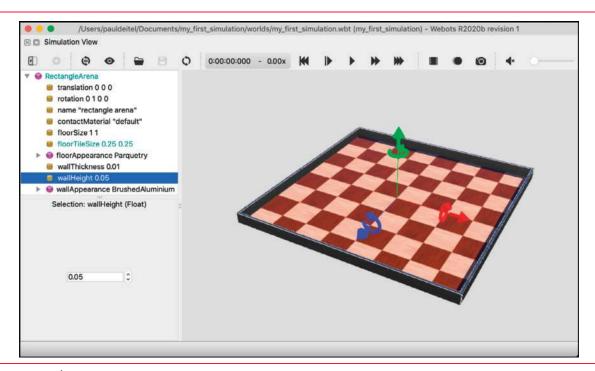


Fig. 7.26 | RectangleArena after changing the checkerboard floor pattern's square size and reducing the wall height.

Step 4: Add WoodenBox Obstacles

The Webots environment comes with almost **800 predefined objects and robots**—called **PROTO nodes**. A PROTO node describes a complex object or robot that you can add to your simulations. The wide variety of PROTO nodes enables you to create realistic 3D simulations of real-world environments. You also can create your own PROTO nodes.

In this step, you'll add a WoodenBox PROTO node using the Add a node dialog (Fig. 7.27). When you select a PROTO node in the dialog, it shows a brief node description, provides a link to the node's more detailed online documentation and shows the node's licensing information (with a link for more license information). Browse through the Add a node dialog to get a sense of the wide variety of animate (robots and vehicles) and inanimate (walls, buildings, furniture, plants, etc.) PROTO nodes you can use in your simulations.

Next, you'll size the WoodenBox and move it. Then you'll make two copies and move them to other locations within the RectangleArena. When this step asks you to copy-and-paste a WoodenBox, the new one will have the same size and location as the one you copied. Hold the *Shift* key and drag the new one to a different location to see it. We found it easier to copy nodes by selecting them in the scene tree. When you complete this step, your world should appear similar to Fig. 7.28. The last WoodenBox you moved will be selected in your world.¹⁵

^{15. &}quot;Webots User Guide R2020b revision 2 — The 3D Window." Accessed December 13, 2020. https://cyberbotics.com/doc/guide/the-3d-window#moving-a-solid-object.

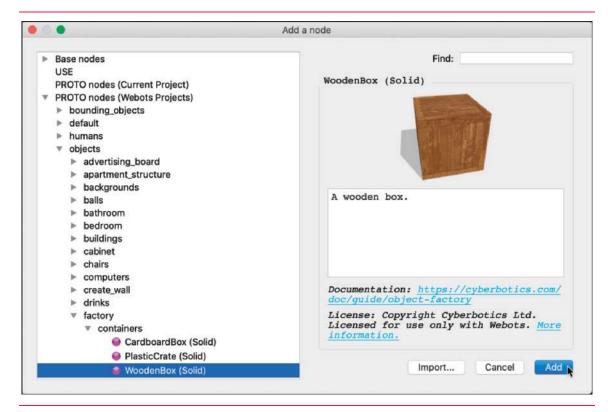


Fig. 7.27 | Selecting a WoodenBox PROTO node in the Add a node dialog.

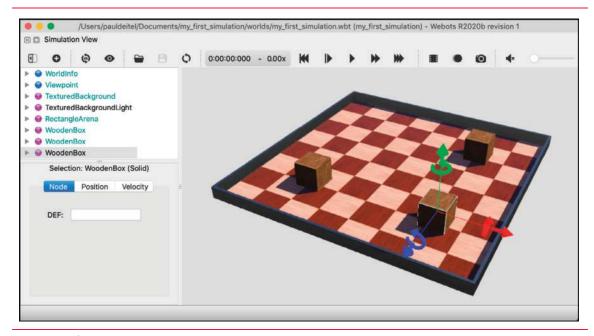


Fig. 7.28 Virtual world after creating and positioning three WoodenBox objects.

Step 5: Add a Robot to Your Virtual World

In this step, you'll use the Add a node dialog to add an e-puck robot to the simulation (Fig. 7.29). When you select the E-puck (Robot) node, the dialog displays a description of the robot, the robot's website (http://www.e-puck.org), the robot's Webots documentation link and the robot's license information.

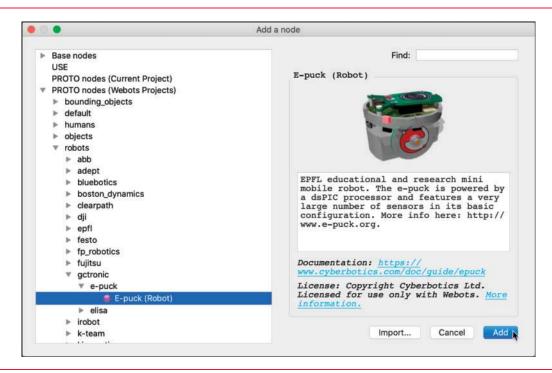


Fig. 7.29 | Adding an e-puck robot PROTO node to the simulation.

The e-puck is preconfigured to move forward, rotating left to change direction if it collides with an obstacle, such as a WoodenBox or a wall. Though small, an e-puck robot actually is loaded with technology, including distance sensors, which can be used to program it to avoid collisions entirely. In Webots Tutorial 4, you'll learn how to avoid obstacles using these distance sensors.

A robot's behavior is specified by its controller. The default e-puck robot controller we just described is named e-puck_avoid_obstacles (you can view this code in the text editor, by selecting Tools > Text Editor). Studying existing bundled controllers is a great way to learn more about controlling Webots robots. To complete this step, you'll run the e-puck robot's default controller (Fig. 7.30). We clicked the world's background area to deselect all the simulation elements.

Step 6: Playing with Physics

The Webots simulator has a physics engine that enables objects to act and interact as they would in the real world. Some physics options you can configure include density, mass, inertia, friction and bounce. For more, visit:

https://cyberbotics.com/doc/reference/physics

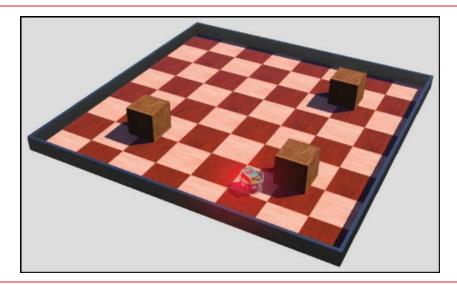


Fig. 7.30 | Simulation with an e-puck robot roaming the **RectangleArena**.

In this step, you'll use your mouse to **apply a force to the e-puck robot**. After you do this, your world should appear similar to Fig. 7.31, with the red arrow indicating the force's direction. When you perform this step, you might accidentally tip over the robot if you apply too much force—as shown in Fig. 7.31. Of course, robots can tip over in the real world, too. To fix this in the simulation, click the **Reset Simulation** button, which will restore the simulation to the point of your most recent save.

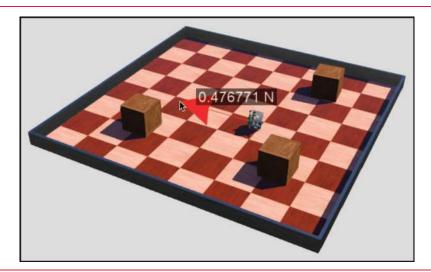


Fig. 7.31 | Manually applying a force to the e-puck robot. In this case, too much force was applied, tipping over the robot.

The WoodenBox obstacles you created in **Step 4** are stuck to the floor by default and do not move when the e-puck bumps into them. You'll see in this step that setting the WoodenBoxes' masses enables them to respond to force. The smaller a WoodenBox's mass,

the more it will move when the e-puck robot collides with it. The tutorial recommends setting the WoodenBoxes' masses to 0.2 kilograms. Try setting each WoodenBox's mass to smaller and larger values to see how these masses affect the physics interactions.

Step 7: Decrease the World's Time Step

Throughout a simulation, Webots keeps track of your simulation's virtual time. The basic time step is a value in milliseconds. Throughout the simulation, when virtual time increases by the basic time step, Webots performs its physics calculations:¹⁶

- Larger basic-time-step values decrease the physics-calculation frequency. This enables simulations to run faster because they perform fewer calculations; however, this can make physics interactions, such as collisions between objects, less accurate, and the simulation can feel clunky.
- Smaller values increase the physics-calculation frequency, making physics calculations more accurate. This causes simulations to run slower because they perform more calculations, but movements may appear smoother.

When you created this simulation's files, Webots set the basic time step to 32 milliseconds. In this step, you'll decrease the basic time-step value to 16 milliseconds. For tips on Webots simulation speed and performance, see:

https://www.cyberbotics.com/doc/guide/speed-performance

Step 8: Create a C Source-Code File for Your Robot's Controller

In this step, you'll replace the default e-puck_avoid_obstacles controller with a new custom controller. Many robots can use each controller you create, but each robot may have only one controller at a time. Figures 7.32–7.35 show the steps you'll go through after selecting Wizards > New Robot Controller.... These steps will create a new C source-code file for your custom controller and open it in the code editor at the Webots environment's right side:

- In the New controller creation step (Fig. 7.32), you'll simply click Continue.
- In the Language selection step (Fig. 7.33), ensure that **C** is selected, then click **Continue.**
- In the Name selection step (Fig. 7.34), change the default custom controller name to epuck_go_forward from my_controller and click Continue.
- The Conclusion step (Fig. 7.35) shows all the folders and files the wizard will generate for your controller.

^{16. &}quot;Webots Reference Manual R2020b revision 2: WorldInfo." Accessed December 12, 2020. https://www.cyberbotics.com/doc/reference/worldinfo.



Fig. 7.32 | Initial Create a new robot controller wizard window.

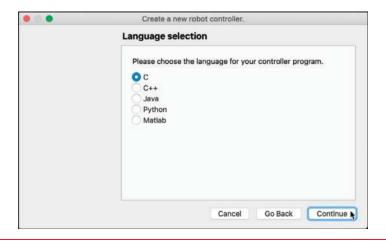


Fig. 7.33 | Selecting the C programming language for your custom robot controller.

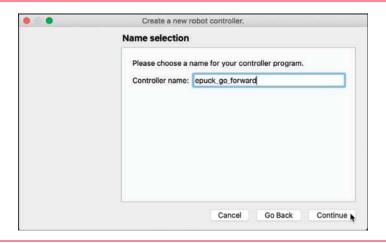


Fig. 7.34 | Changing the default custom controller name to epuck_go_forward.

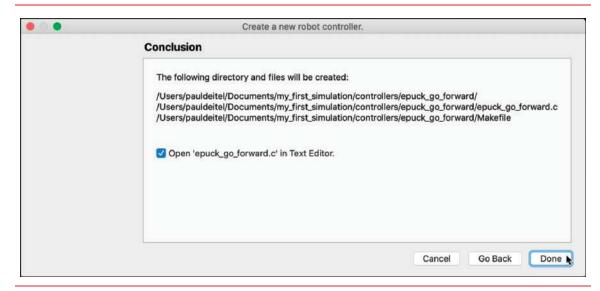


Fig. 7.35 Summary of the folders and files the wizard will generate for your custom controller.

Step 9: Modify Your Controller's Code to Move the Robot Forward

The new controller you created in **Step 8** contains the basic framework of a simple controller. In this step, you'll add code to this file that will make the e-puck move forward a short distance. The tutorial does not specify precisely where each statement should be added to the controller code, so make the following changes to your controller's source-code file:

1. Add the following #include before main:

```
#include <webots/motor.h>
```

2. Next, you'll use the Webots wb_robot_get_device function¹⁷ to get the devices that represent the e-puck robot's left- and right-wheel motors. Add the following code in main after the call to wb_robot_init:

```
// get the motor devices
WbDeviceTag left_motor =
   wb_robot_get_device("left wheel motor");
WbDeviceTag right_motor =
   wb_robot_get_device("right wheel motor");
```

3. Finally, you'll use the Webots wb_motor_set_position function¹⁸ to move the robot a short distance. Add the following code after the calls to wb_robot_get_device and before the while loop:

```
wb_motor_set_position(left_motor, 10.0);
wb_motor_set_position(right_motor, 10.0);
```

^{17. &}quot;Webots Reference Manual R2020b revision 2—Robot." Accessed December 13, 2020. https://cyberbotics.com/doc/reference/robot#wb_robot_get_device.

^{18. &}quot;Webots Reference Manual R2020b revision 2—Motor." Accessed December 13, 2020. https://cyberbotics.com/doc/reference/motor#wb_motor_set_position.

Experiment with different values for wb_motor_set_position's second argument to get a sense of how they affect the distance traveled. Also try using different values in the two calls to wb_motor_set_position so that the two wheels do not rotate the same amount.

Step 10: Modify Your Controller's Code to Change the Robot's Speed In this final step, you'll modify your controller code to specify the wheels' speeds. Make the following changes to your controller's source-code file:

1. Speeds in Webots use radians for rotational motors, such as those used in wheels; otherwise, speeds use meters per second. Add the following #define directive after #include <webots/robot.h> to define the robot's maximum wheel-rotation speed in radians (6.28 is 2π radians):

```
#define MAX_SPEED 6.28
```

2. Modify your two calls to the wb_motor_set_position function, replacing their second arguments with the Webots constant INFINITY, so the wheels spin continuously (at the speed you'll set momentarily) throughout the simulation:

```
// set wheels to spin continuously
wb_motor_set_position(left_motor, INFINITY);
wb_motor_set_position(right_motor, INFINITY);
```

3. Finally, you'll use the Webots function wb_motor_set_velocity function¹⁹ to specify the wheel-rotation speeds in radians per second. Add the following code after the calls to wb_robot_get_device and before the while loop:

```
// set up the motor speeds at 10% of the MAX_SPEED
wb_motor_set_velocity(left_motor, 0.1 * MAX_SPEED);
wb_motor_set_velocity(right_motor, 0.1 * MAX_SPEED);
```

Experiment with different values for wb_motor_set_velocity's second argument to see how they affect the robot's speed. Try using different values in the two calls so that the two wheels do not rotate in unison. Be careful how you set these values—they could cause the robot to spin around in circles.

Additional Tutorials

Once you complete **Tutorial 1**, you may wish to continue with Webots **Tutorials 2–8**. **Tutorials 2–4** perform various modifications to the world you just created:

- In Tutorial 2, you'll add a ball to your world. You'll learn more about node
 types and how to configure physics options that enable the ball to roll in the
 simulation.
- In Tutorial 3, you'll learn how to improve your simulation's graphics with lighting effects and textures.

^{19. &}quot;Webots Reference Manual R2020b revision 2—Motor." Accessed December 13, 2020. https://cyberbotics.com/doc/reference/motor#wb_motor_set_velocity.

• In **Tutorial** 4, you'll create a more elaborate controller that enables the e-puck to use **its distance sensors to avoid the obstacles** you created previously.

Challenge: Once you complete Tutorial 4, try building a maze in your world and see if you can program the e-puck robot to traverse the maze. The following page lists several algorithms for finding a maze's exit:

https://en.wikipedia.org/wiki/Maze_solving_algorithm

Tutorials 5-7 dive into more advanced features:

- In Tutorial 5, you'll learn more about physics in Webots.
- In **Tutorial 6**, you'll work with a **four-wheeled robot** and learn more about sensors.
- In Tutorial 7, you'll create your own PROTO node.

In the advanced **Tutorial 8**, you'll learn about working with Webots nodes from the webots_ros. ROS is the Robot Operating System²⁰—a framework for writing robot software. If you're going to do this tutorial, Webots recommends that you first learn more about ROS in the tutorials at

http://wiki.ros.org/ROS/Tutorials

Mastering Webots Tutorials 2–8 will be a "resume-worthy" accomplishment.

7.32 (Challenge Project: Webots Tortoise-and-Hare-Race Case Study) For this challenging exercise, we recommend that you first complete the Webots tutorials 2–7 mentioned at the end of Exercise 7.31. In Exercise 5.54, you simulated the Tortoise and the Hare race. Now that you're familiar with the Webots 3D robotics simulator, let your imagination run wild with Webots' amazing capabilities. Recreate the race using two robots from the dozens available in Webots. Consider using a small slow one for the tortoise (such as the e-puck from Exercise 7.31) and a larger fast one for the hare (such as the Boston Dynamics Spot^{21,22} robot). Recall from the **Webots Guided Tour** that there are many other environments in which your robots may roam. Consider copying an existing terrain environment with objects like grass, flowers, trees and hills in which to race your robots.

^{20. &}quot;About ROS." Accessed December 13, 2020. https://www.ros.org/about-ros/.

^{21. &}quot;Webots User Guide—Boston Dynamics' Spot." Accessed December 31, 2020. https://www.cyberbotics.com/doc/guide/spot.

^{22. &}quot;Spot." Accessed December 31, 2020. https://www.bostondynamics.com/spot.

Characters and Strings



Objectives

In this chapter, you'll:

- Use the functions of the character-handling library (<ctype.h>).
- Use the string-conversion functions of the general utilities library (<stdlib.h>).
- Use the string and character input/output functions of the standard input/output library (<stdio.h>).
- Use the string-processing functions of the stringhandling library (<string.h>).
- Use the memory-processing functions of the stringhandling library (<string.h>).

- **8.1** Introduction
- **8.2** Fundamentals of Strings and Characters
- **8.3** Character-Handling Library
 - 8.3.1 Functions isdigit, isalpha, isalnum and isxdigit
 - 8.3.2 Functions islower, isupper, tolower and toupper
 - 8.3.3 Functions isspace, iscntrl,
 ispunct, isprint and
 isgraph
- **8.4** String-Conversion Functions
 - 8.4.1 Function strtod
 - 8.4.2 Function strtol
 - 8.4.3 Function strtoul
- **8.5** Standard Input/Output Library Functions
 - 8.5.1 Functions fgets and putchar
 - 8.5.2 Function getchar
 - 8.5.3 Function sprintf
 - 8.5.4 Function sscanf
- **8.6** String-Manipulation Functions of the String-Handling Library
 - 8.6.1 Functions strcpy and strncpy
 - 8.6.2 Functions strcat and strncat

- **8.7** Comparison Functions of the String-Handling Library
- **8.8** Search Functions of the String-Handling Library
 - 8.8.1 Function strchr
 - 8.8.2 Function strcspn
 - 8.8.3 Function strpbrk
 - 8.8.4 Function strrchr
 - 8.8.5 Function strspn
 - 8.8.6 Function strstr
 - 8.8.7 Function strtok
- **8.9** Memory Functions of the String-Handling Library
 - 8.9.1 Function memcpy
 - 8.9.2 Function memmove
 - 8.9.3 Function memcmp
 - 8.9.4 Function memchr
 - 8.9.5 Function memset
- **8.10** Other Functions of the String-Handling Library
 - 8.10.1 Function strerror
 - 8.10.2 Function strlen
- **8.11** Secure C Programming

Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises |
Special Section: Advanced String-Manipulation Exercises | A Challenging String-Manipulation Project | Pqyoaf X Nylfomigrob Qwbbfmh Mndogvk: Rboqlrut yua Boklnxhmywex |
Secure C Programming Case Study: Public-Key Cryptography

8.1 Introduction

This chapter introduces the C standard library functions that help you process characters, strings, lines of text and blocks of memory. The chapter discusses the techniques used to develop editors, word processors, page-layout software and other kinds of text-processing software. The text manipulations performed by formatted input/output functions like printf and scanf can be implemented using the functions this chapter presents.

8.2 Fundamentals of Strings and Characters

Characters are the fundamental building blocks of your programs. Every program is composed of characters that—when grouped together meaningfully—the computer interprets as a series of instructions used to accomplish a task. A program may contain character constants—each is an int value represented as a character in single quotes. A character constant's value is that character's integer value in the machine's character

set. For example, 'z' represents the letter z's integer value, and '\n' represents a newline's integer value.

A string is a series of characters treated as a single unit. A string may include letters, digits and various special characters such as +, -, *, / and \$. String literals, or string constants, are written in double quotation marks as follows:

```
"John Q. Doe"
                            (a name)
"99999 Main Street"
                            (a street address)
"Waltham, Massachusetts"
                            (a city and state)
                            (a telephone number)
"(201) 555-1212"
```

Strings Are Null Terminated

Every string must end with the null character ('\0'). Printing a "string" that does not \(\infty \mathbb{ERR} \) contain a terminating null character is a logic error. The results of this are undefined. On some systems, printing will continue past the end of the "string" until a null character is encountered. On others, your program will terminate prematurely (i.e., "crash") and indicate a "segmentation fault" or "access violation" error.

Strings and Pointers

You access a string via a *pointer* to its first character. A string's "value" is the address of its first character. Thus, in C, it's appropriate to say that a string is a pointer to the string's first character. This is just like arrays, because strings are simply arrays of characters.

Initializing char Arrays and char * Pointers

You can initialize a character array or a char * variable with a string. The definitions

```
char color[] = "blue";
const char *colorPtr = "blue";
```

initialize color and colorPtr to the string "blue". The first definition creates a 5-element array color containing the *modifiable* characters 'b', 'l', 'u', 'e' and '\0'. The second definition creates the pointer variable colorPtr that points to the letter 'b' in "blue", which is not modifiable.

The color array definition also can be written as

```
char color[] = {'b', 'l', 'u', 'e', '\0'};
```

The preceding definition automatically determines the array's size based on its number of initializers (5). When storing a string in a char array, the array must be large enough to store the string *and* its terminating null character. Not allocating sufficient (X) ERR space in a character array to store the null character that terminates a string is an error. C allows you to store strings of any length. If a string is longer than the char array in which you store it, characters beyond the array's end may overwrite other data in memory.

String Literals Should Not Be Modified



SE A The C standard indicates that a string literal is immutable—that is, not modifiable. If you might need to modify a string, it must be stored in a character array.

Reading a String with scanf

Function scanf can read a string and store it in a char array. Assume we have a char array word containing 20 elements. You can read a string into the array with

```
scanf("%19s", word);
```

Since word is an array, the array name is a pointer to the array's first element. So, the & that we typically use with scanf's arguments is not required.

Recall from Section 6.5.4 that scanf reads characters until it encounters a space, tab, newline or end-of-file indicator. The field width 19 in the preceding statement ensures that scanf reads a *maximum* of 19 characters, saving the last array element for the string's terminating null character. This prevents scanf from writing characters into memory beyond the array's last element.

Without the field width 19 in the conversion specification %19s, the user input could exceed 19 characters and overwrite other data in memory. If so, your program might crash, or overwrite other data in memory. So, always use a field width when reading strings with scanf. (For reading input lines of arbitrary length, there's a nonstandard—yet widely supported—function readline, usually included in stdio.h.)

Self Check

- (Fill-In) A string is accessed via a ______ to the string's first character. **Answer:** pointer.
- **2** (*True/False*) The following definition initializes the color array to the character string "blue":

```
char color[] = {'b', 'l', 'u', 'e'};
```

Answer: False. Actually, to be a character string, the color array must end with the null character, as in

```
char color[] = {'b', 'l', 'u', 'e', '\0'};
```

3 (True/False) Printing a string that does not contain a terminating null character is a logic error—program execution terminates immediately.

Answer: False. Actually, printing will continue past the end of the string until a null character is encountered.

8.3 Character-Handling Library

The character-handling library (<ctype.h>) contains functions that test and manipulate character data. Each function receives an unsigned char (represented as an int) or EOF as an argument. As we discussed in Chapter 4, characters are often manipulated as integers because a character in C is a one-byte integer. EOF's value is typically -1. The following table summarizes the character-handling library functions.

Prototype	Function description
<pre>int isblank(int c);</pre>	Returns a true value if c is a blank character that separates words in a line of text; otherwise, it returns 0 (false).
<pre>int isdigit(int c);</pre>	Returns a true value if c is a digit; otherwise, it returns 0 (false).
<pre>int isalpha(int c);</pre>	Returns a true value if c is a letter; otherwise, it returns 0 (false).
<pre>int isalnum(int c);</pre>	Returns a true value if c is a digit or a letter; otherwise, it returns 0 (false).
<pre>int isxdigit(int c);</pre>	Returns a true value if c is a hexadecimal digit character; otherwise, it returns 0 (false). (See online Appendix E for a detailed explanation of binary numbers, octal numbers, decimal numbers and hexadecimal numbers.)
<pre>int islower(int c);</pre>	Returns a true value if c is a lowercase letter; otherwise, it returns 0 (false).
<pre>int isupper(int c);</pre>	Returns a true value if c is an uppercase letter; otherwise, it returns 0 (false).
<pre>int tolower(int c);</pre>	If c is an uppercase letter, tolower returns c as a lowercase letter; otherwise, it returns the argument unchanged.
<pre>int toupper(int c);</pre>	If c is a lowercase letter, toupper returns c as an uppercase letter; otherwise, it returns the argument unchanged.
<pre>int isspace(int c);</pre>	Returns a true value if c is a whitespace character—newline ('\n'), space (' '), form feed ('\f'), carriage return ('\r'), horizontal tab ('\t') or vertical tab ('\v')—otherwise, it returns 0 (false).
<pre>int iscntrl(int c);</pre>	Returns a true value if c is a control character—horizontal tab ('\t'), vertical tab ('\v'), form feed ('\f'), alert ('\a'), backspace ('\b'), carriage return ('\r'), newline ('\n') and others—otherwise, it returns 0 (false).
<pre>int ispunct(int c);</pre>	Returns a true value if c is a printing character other than a space, a digit, or a letter—such as \$, #, (,), [,], {, }, ;, : or %—otherwise, it returns 0 (false).
<pre>int isprint(int c);</pre>	Returns a true value if c is a printing character (i.e., a character that's visible on the screen) including a space; otherwise, it returns 0 (false).
<pre>int isgraph(int c);</pre>	Returns a true value if c is a printing character other than a space; otherwise, it returns 0 (false).

8.3.1 Functions isdigit, isalpha, isalnum and isxdigit

Figure 8.1 demonstrates functions **isdigit**, **isalpha**, **isalnum** and **isxdigit**. Function isdigit determines whether its argument is a digit (0–9). Function isalpha determines whether its argument is an uppercase (A–Z) or lowercase letter (a–z). Function isalnum determines whether its argument is an uppercase letter, a lowercase letter or a digit. Function isxdigit determines whether its argument is a hexadecimal digit (A–F, a–f, 0–9).

```
// fig08_01.c
     // Using functions isdigit, isalpha, isalnum, and isxdigit
 3 #include <ctype.h>
 4 #include <stdio.h>
 5
 6
     int main(void) {
         7
 8
 9
             isdigit('#') ? "# is a " : "# is not a ", "digit");
10
         printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n", "According to isalpha:",
11
             isalpha('A') ? "A is a " : "A is not a ", "letter", 'calpha('h') ? "h is a " : "b is not a ", "letter",
12
13
             isalpha('b') ? "b 15 a : b is not a , letter", isalpha('&') ? "& is a " : "& is not a ", "letter",
14
             isalpha('4') ? "4 is a " : "4 is not a ", "letter");
15
16
17
         printf("%s\n%s%s\n%s%s\n%s%s\n\n", "According to isalnum:",
             isalnum('A') ? "A is a " : "A is not a ", "digit or a letter",
isalnum('8') ? "8 is a " : "8 is not a ", "digit or a letter",
isalnum('#') ? "# is a " : "# is not a ", "digit or a letter");
18
19
20
21
         printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n", "According to isxdigit:",
    isxdigit('F') ? "F is a " : "F is not a ", "hexadecimal digit",
    isxdigit('J') ? "J is a " : "J is not a ", "hexadecimal digit",
22
23
24
             isxdigit('7') ? "7 is a " : "7 is not a ", "hexadecimal digit'
25
             isxdigit('f') ? 'f is a : / is not a ', nexadecimal digit',
isxdigit('f') ? "f is a " : "f is not a ", "hexadecimal digit",
isxdigit('f') ? "f is a " : "f is not a ", "hexadecimal digit");
26
27
28 }
According to isdigit:
8 is a digit
# is not a digit
According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter
According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter
According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

Fig. 8.1 | Using functions isdigit, isalpha, isalnum and isxdigit.

Figure 8.1 uses the conditional operator (?:) to determine whether the string " is a " or the string " is not a " should be printed in the output for each character tested. For example, the expression

```
isdigit('8') ? "8 is a " : "8 is not a "
```

indicates that if '8' is a digit, the string "8 is a " is printed, and if '8' is not a digit (i.e., isdigit returns 0), the string "8 is not a " is printed.

8.3.2 Functions is lower, isupper, to lower and toupper

Figure 8.2 demonstrates functions **islower**, **isupper**, **tolower** and **toupper**. Function islower determines whether its argument is a lowercase letter (a–z). Function isupper determines whether its argument is an uppercase letter (A–Z). Function tolower converts an uppercase letter to a lowercase letter and returns the lowercase letter. If the argument is not an uppercase letter, tolower returns the argument unchanged. Function toupper converts a lowercase letter to an uppercase letter and returns the uppercase letter. If the argument is not a lowercase letter, toupper returns the argument unchanged.

```
| // fig08_02.c
 2 // Using functions islower, isupper, tolower and toupper
 3 #include <ctype.h>
 4 #include <stdio.h>
 5
 6
       int main(void) {
            printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n", "According to islower:",
 7
                 islower('p') ? "p is a " : "p is not a ", "lowercase letter",
islower('P') ? "P is a " : "P is not a ", "lowercase letter",
islower('5') ? "5 is a " : "5 is not a ", "lowercase letter",
islower('!') ? "! is a " : "! is not a ", "lowercase letter");
 8
 9
10
11
12
            printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n", "According to isupper:",
13
                 isupper('D') ? "D is an " : "D is not an ", "uppercase letter",
isupper('d') ? "d is an " : "d is not an ", "uppercase letter",
isupper('8') ? "8 is an " : "8 is not an ", "uppercase letter",
isupper('$') ? "$ is an " : "$ is not an ", "uppercase letter");
14
15
16
17
18
            printf("%s%c\n%s%c\n%s%c\n",
19
                 "u converted to uppercase is ", toupper('u'),
"7 converted to uppercase is ",
"$ converted to uppercase is ",
"L converted to lowercase is ",
tolower('L'));
20
21
22
23
24 }
According to islower:
p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter
```

Fig. 8.2 | Using functions islower, isupper, tolower and toupper. (Part I of 2.)

```
According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter
u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is 1
```

Fig. 8.2 Using functions islower, isupper, tolower and toupper. (Part 2 of 2.)

8.3.3 Functions isspace, iscntrl, ispunct, isprint and isgraph

Figure 8.3 demonstrates functions isspace, iscntrl, ispunct, isprint and isgraph. Function isspace determines whether a character is one of the following whitespace characters: space (''), form feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t') or vertical tab ('\v'). Function iscntrl determines whether a character is one of the following control characters: horizontal tab ('\t'), vertical tab ('\v'), form feed ('\f'), alert ('\a'), backspace ('\b'), carriage return ('\r') or newline ('\n'). Function ispunct determines whether a character is a printing character other than a space, a digit or a letter, such as \$, #, (,), [,], {, }, ;, : or %. Function isprint determines whether a character can be displayed on the screen (including the space character). Function isgraph is the same as isprint, except that the space character is not included.

```
I // fig08_03.c
    // Using functions isspace, iscntrl, ispunct, isprint and isgraph
 2
 3 #include <ctype.h>
 4 #include <stdio.h>
 5
 6
     int main(void) {
         printf("\%s\n\%s\%s\%s\n\%s\%s\%s\n'', "According to is space:",
 7
              "Newline", isspace('\n') ? " is a " : " is not a ",
 8
 9
              "whitespace character",
              "Horizontal tab", isspace('\t')? " is a " : " is not a ",
10
              "whitespace character",
11
              isspace('%') ? "% is a " : "% is not a ", "whitespace character");
12
13
14
         printf("%s\n%s%s%s\n\n", "According to iscntrl:",
              "Newline", iscntrl('\n') ? " is a " : " is not a ".
15
              "control character",
16
              iscntrl('$') ? "$ is a " : "$ is not a ", "control character");
17
18
        printf("%s\n%s%s\n%s%s\n%s%s\n\n", "According to ispunct:",
    ispunct(';') ? "; is a " : "; is not a ", "punctuation character",
    ispunct('Y') ? "Y is a " : "Y is not a ", "punctuation character",
    ispunct('#') ? "# is a " : "# is not a ", "punctuation character");
19
20
21
22
```

Fig. 8.3 Using functions isspace, iscntrl, ispunct, isprint and isgraph. (Part 1 of 2.)

```
23
      24
25
          "Alert", isprint('\a') ? " is a " : " is not a ",
26
27
          "printing character");
28
      printf("%s\n%s%s\n%s%s\n", "According to isgraph:",
29
          isgraph('Q') ? "Q is a " : "Q is not a ",
30
          "printing character other than a space",
31
          "Space", isgraph(' ') ? " is a " : " is not a ",
32
          "printing character other than a space");
33
34 }
According to isspace:
Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character
According to iscntrl:
Newline is a control character
$ is not a control character
According to ispunct:
; is a punctuation character
Y is not a punctuation character
# is a punctuation character
According to isprint:
$ is a printing character
Alert is not a printing character
According to isgraph:
Q is a printing character other than a space
Space is not a printing character other than a space
```

Fig. 8.3 Using functions isspace, iscntrl, ispunct, isprint and isgraph. (Part 2 of 2.)

Self Check

- **I** (*Multiple Choice*) Which functions is described by "Returns a true value if the argument character is a digit or a letter; otherwise, returns 0 (false)"?
 - a) isalnum.
 - b) isdigit.
 - c) isalpha.
 - d) isxdigit.

Answer: a.

2 (*Code*) What does the following printf print?

```
printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n", "According to isalpha:",
   isalpha('X') ? "X is a " : "X is not a ", "letter",
   isalpha('m') ? "m is a " : "m is not a ", "letter",
   isalpha('$') ? "$ is a " : "$ is not a ", "letter",
   isalpha('7') ? "7 is a " : "7 is not a ", "letter");
```

Answer:

```
According to isalpha:
X is a letter
m is a letter
$ is not a letter
7 is not a letter
```

8.4 String-Conversion Functions

This section presents the string-conversion functions from the general utilities library (<std1ib.h>). These functions convert strings of digits to integer and floating-point values. The following table summarizes the string-conversion functions. The C standard also includes strtoll and strtoull for converting strings to long long int and unsigned long long int, respectively.

```
Function prototype Function description

double strtod(const char *nPtr, char **endPtr);

Converts the string nPtr to double.

long strtol(const char *nPtr, char **endPtr, int base);

Converts the string nPtr to long.

unsigned long strtoul(const char *nPtr, char **endPtr, int base);

Converts the string nPtr to unsigned long.
```

8.4.1 Function strtod

Function strtod (Fig. 8.4) converts a sequence of characters representing a floating-point value to double. The function returns 0 if it's unable to convert part of its first argument to double. The function receives two arguments—a string (char *) and a pointer to a string (char **). The string argument contains the character sequence to be converted to double. Whitespace characters at the beginning of the string are ignored. The function uses the char ** argument to aim a char * in the caller (stringPtr) at the first character after the converted portion of the string. If nothing can be converted, the function aims the pointer at the beginning of the string. Line 10 assigns d the double value converted from string and aims stringPtr at the % in string.

```
// fig08_04.c
// Using function strtod
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    const char *string = "51.2% are admitted";
    char *stringPtr = NULL;
```

Fig. 8.4 Using function strtod. (Part I of 2.)

```
double d = strtod(string, &stringPtr);

printf("The string \"%s\" is converted to the\n", string);
printf("double value %.2f and the string \"%s\"\n", d, stringPtr);
}

The string "51.2% are admitted" is converted to the double value 51.20 and the string "% are admitted"
```

Fig. 8.4 Using function strtod. (Part 2 of 2.)

8.4.2 Function strtol

Function **strto1** (Fig. 8.5) converts to long int a sequence of characters representing an integer. The function returns 0 if it's unable to convert any portion of its first argument to long int. The function's three arguments are a string (char *), a pointer to a string and an integer. This function works identically to strtod, but the third argument specifies the *base* of the value being converted.

```
I // fig08_05.c
2 // Using function strtol
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(void) {
7
       const char *string = "-1234567abc";
       char *remainderPtr = NULL;
8
9
       long x = strtol(string, &remainderPtr, 0);
10
11
       printf("%s\"%s\"\n%s%ld\n%s\"%s\"\n%s%ld\n",
12
13
          "The original string is ", string,
          "The converted value is ", x,
14
          "The remainder of the original string is ", remainderPtr,
15
          "The converted value plus 567 is ", x + 567);
17 }
The original string is "-1234567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000
```

Fig. 8.5 | Using function strtol.

Line 10 assigns x the long value converted from string and aims remainderPtr at the "a" in string. Using NULL for the second argument causes the *remainder of the string to be ignored*. The third argument, 0, indicates that the value to convert can be in octal (base 8), decimal (base 10) or hexadecimal (base 16) format. The base can be

specified as 0 or any value between 2 and 36. Integer representations from base 11 to base 36 use the letters A–Z to represent the integer values 10–35. For example, hexadecimal values can consist of the digits 0–9 and the characters A–F.

8.4.3 Function strtoul

Function **strtoul** (Fig. 8.6) converts to unsigned long int a sequence of characters representing an unsigned long int value. The function works identically to function strtol. Line 10 assigns x the unsigned long int value converted from string and aims remainderPtr at the "a" in string. The third argument, 0, indicates that the value to convert can be in octal, decimal or hexadecimal format.

```
I // fig08_06.c
   // Using function strtoul
3 #include <stdio.h>
4 #include <stdlib.h>
   int main(void) {
7
       const char *string = "1234567abc";
8
       char *remainderPtr = NULL;
9
10
       unsigned long int x = strtoul(string, &remainderPtr, 0);
11
       printf("%s\"%s\"\n%s%lu\n%s\"%s\"\n%s%lu\n",
12
          "The original string is ", string,
13
          "The converted value is ", x,
          "The remainder of the original string is ", remainderPtr,
15
16
          "The converted value minus 567 is ", x - 567);
17 }
The original string is "1234567abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000
```

Fig. 8.6 | Using function strtoul.

Self Check

- **I** (*Discussion*) Why would a function's parameter list contain a char ** parameter? **Answer:** A char ** typically is a pointer to a char * pointer in the caller. A called function uses such a pointer to receive a char * by reference to modify it in the caller—for example, to aim it at another string. This is an example of a pointer to a pointer.
- 2 (Multiple Choice) Which of the following statements about function strto1 is false?
 - a) It converts to long int a sequence of characters representing an integer, or it returns 0 if it's unable to convert any portion of its first argument to long int.

^{1.} See online Appendix E for a detailed explanation of the octal, decimal and hexadecimal number systems.

- b) strto1's three arguments are a string (char *), a pointer to a string (char **) and an integer.
- c) The string argument contains the character sequence to convert to long—any whitespace characters at the beginning of the string are ignored.
- d) The function uses the char ** argument to give the caller access to the numeric portion of the string being converted.

Answer: d) is *false*. Actually, the function uses the char ** argument to modify a char * in the caller to point to the location of the first character *after* the string's converted portion. If nothing is converted, the function modifies the char * to point to the entire string.

8.5 Standard Input/Output Library Functions

This section presents the standard input/output (<stdio.h>) library's character- and string-manipulation functions, which we summarize in the following table.

Function prototype	Function description	
<pre>int getchar(void);</pre>	Returns the next character from the standard input as an integer.	
<pre>char *fgets(char *s, int n, FILE *stream);</pre>		
	Reads characters from the specified stream into the array s until a newline or end-of-file character is encountered, or until n - 1 bytes are read. This chapter uses the stream stdin—the standard input stream—to read characters from the keyboard. A terminating null character is appended to the array. Returns the string that was read into s. If a newline is encountered, it's included in the stored string.	
<pre>int putchar(int c);</pre>	Prints the character stored in c and returns it as an integer.	
<pre>int puts(const char *s);</pre>	Prints the string s followed by a newline character. Returns a nonzero integer if successful, or EOF if an error occurs.	
<pre>int sprintf(char *s, const char *format,);</pre>		
	Equivalent to printf, but the output is stored in the array s instead of printed on the screen. Returns the number of characters written to s, or EOF if an error occurs.	
<pre>int sscanf(char *s, const char *format,);</pre>		
	Equivalent to scanf, but the input is read from the array s rather than from the keyboard. Returns the number of items successfully read by the function, or EOF if an error occurs.	

8.5.1 Functions fgets and putchar

Figure 8.7 uses functions **fgets** and **putchar** to read a line of text from the standard input (keyboard) and recursively output the line's characters in reverse order. Line 12 uses **fgets** to read characters into its char array argument until it encounters a newline or the end-of-file indicator, or until the maximum number of characters is read.

The maximum number of characters is one fewer than fgets's second argument. The third argument is the stream from which to read characters—in this case, the standard input stream (stdin). When reading terminates, fgets appends a null character ('\0') to the array. Function putchar (line 27) prints its character argument.

```
I // fig08_07.c
 2 // Using functions fgets and putchar
 3 #include <stdio.h>
 4 #define SIZE 80
 5
   void reverse(const char * const sPtr);
 7
 8
   int main(void) {
       char sentence[SIZE] = "";
 9
10
       puts("Enter a line of text:");
11
12
       fgets(sentence, SIZE, stdin); // read a line of text
13
       printf("\n%s", "The line printed backward is:");
14
15
       reverse(sentence);
       puts("");
16
17
   }
18
   // recursively outputs characters in string in reverse order
19
20 void reverse(const char * const sPtr) {
       // if end of the string
21
       if ('\0' == sPtr[0]) { // base case
22
23
          return;
24
       else { // if not end of the string
25
          reverse(&sPtr[1]); // recursion step
26
27
          putchar(sPtr[0]); // use putchar to display character
28
       }
29 }
Enter a line of text:
Characters and Strings
The line printed backward is:
sgnirtS dna sretcarahC
```

Fig. 8.7 | Using functions fgets and putchar.

Function reverse

The program calls the recursive function reverse² to print the line of text backward. If the array's first character is the null character '\0', reverse returns. Otherwise, reverse calls itself recursively with the subarray's address beginning at element sPtr[1]. Line 27 outputs the character at sPtr[0] when the recursive call completes.

^{2.} We use recursion here for demonstration purposes. It's usually more efficient to use a loop to iterate from a string's last character (the one at the position one less than the string's length) to its first character (the one at position 0).

The order of the two statements in lines 26 and 27 causes reverse to walk to the string's terminating null character *before* displaying any characters. As the recursive calls complete, the characters are output in reverse order.

8.5.2 Function getchar

Figure 8.8 uses functions **getchar** to read one character at a time from the standard input into the character array sentence, then uses puts to display the characters as a string. Function getchar reads a character from the standard input and returns the character as an integer. Recall from Section 4.6 that an integer is returned to support the end-of-file indicator. As you know, puts takes a string as an argument and displays the string followed by a newline character. The program stops inputting characters when 79 characters have been read or when getchar reads a newline character. Line 18 appends a null character to sentence to terminate the string. Then line 21 uses puts to display sentence.

```
// fig08_08.c
 2 // Using function getchar
 3 #include <stdio.h>
 4 #define SIZE 80
 6
   int main(void) {
 7
       int c = 0; // variable to hold character input by user
       char sentence[SIZE] = "";
 8
 9
       int i = 0;
10
       puts("Enter a line of text:");
11
12
       // use getchar to read each character
13
       while ((i < SIZE - 1) && (c = getchar()) != '\n') {
14
15
          sentence[i++] = c;
16
17
       sentence[i] = '\0'; // terminate string
18
19
20
       puts("\nThe line entered was:");
       puts(sentence); // display sentence
21
22
Enter a line of text:
This is a test.
The line entered was:
This is a test.
```

Fig. 8.8 Using function **getchar**.

8.5.3 Function sprintf

Figure 8.9 uses function **sprintf** to print formatted data into char array s. The function uses the same conversion specifications as printf (see Chapter 9 for a detailed

discussion of formatting). The program inputs an int value and a double value to be formatted and printed to array s. Array s is the first argument of sprintf.

```
I // fig08_09.c
2
   // Using function sprintf
3 #include <stdio.h>
4 #define SIZE 80
5
6
   int main(void) {
7
       int x = 0;
8
       double y = 0.0;
9
       puts("Enter an integer and a double:");
10
       scanf("%d%1f", &x, &y);
11
12
       char s[SIZE] = {'\0'}; // create char array
13
       sprintf(s, "integer:%6d\ndouble:%7.2f", x, y);
14
15
       printf("The formatted output stored in array s is:\n%s\n", s);
16
17
   }
Enter an integer and a double:
298 87.375
The formatted output stored in array s is:
integer:
           298
double: 87.38
```

Fig. 8.9 | Using function sprintf.

8.5.4 Function sscanf

Figure 8.10 demonstrates function **sscanf**, which works like scanf but reads formatted data from a string. The program reads an int and a double from char array s, stores them in x and y, then displays them.

```
I // fig08_10.c
2 // Using function sscanf
3 #include <stdio.h>
5
   int main(void) {
6
       char s[] = "31298 87.375";
       int x = 0;
7
8
       double y = 0;
9
       sscanf(s, "%d%lf", &x, &y);
10
11
       puts("The values stored in character array s are:");
12
       printf("integer:%6d\ndouble:%8.3f\n", x, y);
13
    }
```

Fig. 8.10 Using function **sscanf**. (Part 1 of 2.)

```
The values stored in character array s are: integer: 31298 double: 87.375
```

Fig. 8.10 Using function sscanf. (Part 2 of 2.)



Self Check

- **I** (*Multiple Choice*) Which function is described by "Prints the character stored in its parameter and returns it as an integer"?
 - a) getchar.
 - b) sprintf.
 - c) puts.
 - d) putchar.

Answer: d.

2 (*True/False*) Function getchar reads a character from the standard input and returns it as a char.

Answer: *False.* Actually, getchar returns an int to support the end-of-file indicator, which is -1.

8.6 String-Manipulation Functions of the String-Handling Library

The string-handling library (<string.h>) provides useful functions for:

- manipulating string data (copying strings and concatenating strings),
- comparing strings,
- searching strings for characters and other strings,
- tokenizing strings (separating strings into logical pieces), and
- determining the length of strings.

This section presents the string-handling library's string-manipulation functions, which are summarized in the following table. Other than strncpy, each function appends the null character to its result.

```
Function prototype Function description

char *strcpy(char *s1, const char *s2)

Copies string s2 into array s1 and returns s1.

char *strncpy(char *s1, const char *s2, size_t n)

Copies at most n characters of string s2 into array s1 and returns s1.

char *strcat(char *s1, const char *s2)

Appends string s2 to array s1 and returns s1. String s2's first character overwrites s1's terminating null character.
```

```
Function prototype
                       Function description
char *strncat(char *s1, const char *s2, size_t n)
                       Appends at most n characters of string s2 to array s1 and returns s1.
                       String s2's first character overwrites s1's terminating null character.
```

Functions strncpy and strncat specify a size_t parameter. Function strcpy copies the string in the second argument into the char array in its first argument. You must ensure that the array is large enough to store the string and its terminating null character, which is also copied. Function strncpy is equivalent to strcpy but copies only the specified number of characters. Function strncpy will not copy the terminating null character of its second argument unless the number of characters to be copied is more than the string's length. For example, if "test" is the second argument, a terminating null character is written only if the third argument to strncpy is at least 5 (four characters in "test" plus a terminating null character). If the third argument is larger than 5, some implementations append null characters to the array until the total number of characters specified by the third argument is written. Other implementa-ERR (X) tions stop after writing the first null character. It's a logic error if you do not append a terminating null character to strncpy's first argument when the third argument is less than or equal to the second argument's string length.



8.6.1 Functions strcpy and strncpy

Figure 8.11 uses strcpy to copy the entire string in array x into array y. It uses strncpy to copy the first 14 characters of array x into array z. Line 19 appends a null character ('\0') to array z because the strncpy call does not write a terminating null character—the third argument is less than the second argument's string length.

```
I // fig08_11.c
 2 // Using functions strcpy and strncpy
 3 #include <stdio.h>
 4 #include <string.h>
 5 #define SIZE1 25
 6 #define SIZE2 15
 7
 8
   int main(void) {
        char x[] = "Happy Birthday to You"; // initialize char array x
 9
       char y[SIZE1] = ""; // create char array y
char z[SIZE2] = ""; // create char array z
10
11
12
13
        // copy contents of x into y
        printf("%s%s\n%s%s\n",
14
           "The string in array x is: ", x,
15
           "The string in array y is: ", strcpy(y, x));
16
17
```

Fig. 8.11 Using functions strcpy and strncpy. (Part 1 of 2.)

```
strncpy(z, x, SIZE2 - 1); // copy first 14 characters of x into z
z[SIZE2 - 1] = '\0'; // terminate string in z, because '\0' not copied
printf("The string in array z is: %s\n", z);

The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

Fig. 8.11 Using functions strcpy and strncpy. (Part 2 of 2.)

8.6.2 Functions streat and strncat

Function **strcat** appends its second argument string to the string in its char array first argument, replacing the first argument's null ('\0') character. *You must ensure that the array used to store the first string is large enough to store the first string, the second string and the terminating null character copied from the second string.* Function strncat appends a specified number of characters from the second string to the first string and adds a terminating '\0'. Figure 8.12 demonstrates function strcat and function strncat.

```
// fig08_12.c
 2 // Using functions streat and strncat
 3 #include <stdio.h>
 4 #include <string.h>
 5
 6
   int main(void) {
 7
       char s1[20] = "Happy "; // initialize char array s1
 8
       char s2[] = "New Year "; // initialize char array s2
       char s3[40] = ""; // initialize char array s3 to empty
 9
10
11
       printf("s1 = %s\ns2 = %s\n", s1, s2);
12
13
       // concatenate s2 to s1
14
       printf("strcat(s1, s2) = %s\n", strcat(s1, s2));
15
16
       // concatenate first 6 characters of s1 to s3
17
       printf("strncat(s3, s1, 6) = %s\n", strncat(s3, s1, 6));
18
19
       // concatenate s1 to s3
       printf("strcat(s3, s1) = %s\n", strcat(s3, s1));
20
21
s1 = Happy
s2 = New Year
strcat(s1, s2) = Happy New Year
strncat(s3, s1, 6) = Happy
strcat(s3, s1) = Happy Happy New Year
```

Fig. 8.12 Using functions streat and strncat.

Self Check

- I (*Multiple Choice*) Which of the following statements about functions strcat and strncat is *false*?
 - a) Function streat appends its second argument string to the string in its char array first argument.
 - b) The first character of strcat's second argument is placed immediately after the null ('\0') that terminates the string in the first argument.
 - c) You must ensure that the array containing the first string is large enough to store the first string, the second string and the terminating '\0' copied from the second string.
 - d) Function strncat appends a specified number of characters from the second string to the first string. A terminating '\0' is automatically appended to the result.

Answer: b) is *false*. Actually, the first character of strcat's second argument *replaces* the null ('\0') that terminates the string in the first argument.

2 (*Fill-In*) Function strcpy copies its second argument (a string) into its first argument, which is a character array that must be _____. **Answer:** large enough to store the string, including its terminating null character.

8.7 Comparison Functions of the String-Handling Library

This section presents the string-handling library's string-comparison functions, strcmp and strncmp, which are summarized below.

Function prototype	Function description
int strcmp(const cha	r *s1, const char *s2); Compares the string s1 with the string s2. The function returns 0, less than 0 or greater than 0 if s1 is equal to, less than or greater than s2, respectively.
int strncmp(const ch	Compares up to n characters of the string s1 with the string s2. The function returns 0, less than 0 or greater than 0 if s1 is equal to, less than or greater than s2, respectively.

Figure 8.13 compares three strings using strcmp and strncmp. Function strcmp performs a character-by-character comparison of its two string arguments. The function returns:

- 0 if the strings are equal,
- a *negative value* if the first string is less than the second string, or
- a positive value if the first string is greater than the second string.

Function strncmp is equivalent to strcmp but compares up to a specified number of characters. Function strncmp does *not* compare characters following a null character in a string. The program prints the integer value returned by each function call.

```
// fig08_13.c
 2
    // Using functions strcmp and strncmp
 3 #include <stdio.h>
 4
   #include <string.h>
 5
 6
    int main(void) {
 7
        const char *s1 = "Happy New Year"; // initialize char pointer
 8
        const char *s2 = "Happy New Year"; // initialize char pointer
 9
        const char *s3 = "Happy Holidays"; // initialize char pointer
10
П
        printf("s1 = %s\ns2 = %s\ns3 = %s\n\n\%s\%2d\n\%s\%2d\n\%s\%2d\n\n",
12
            s1, s2, s3,
           "strcmp(s1, s2) = ", strcmp(s1, s2), strcmp(s1, s3), strcmp(s3, s1) = ", strcmp(s3, s1);
13
14
15
16
        printf("%s%2d\n%s%2d\n",
17
            "strncmp(s1, s3, 6) = ", strncmp(s1, s3, 6),
"strncmp(s1, s3, 7) = ",
"strncmp(s3, s1, 7) = ",
strncmp(s3, s1, 7);
18
19
20
21
    }
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays
strcmp(s1, s2) = 0
strcmp(s1, s3) = 1
strcmp(s3, s1) = -1
strncmp(s1, s3, 6) = 0
strncmp(s1, s3, 7) =
strncmp(s3, s1, 7) = -1
```

Fig. 8.13 Using functions strcmp and strncmp.

How Strings Are Compared

To understand what it means for one string to be "greater than" or "less than" another, consider the process of alphabetizing last names. You'd, no doubt, place "Jones" before "Smith" because "J" comes before "S" in the alphabet. But the alphabet is more than just a list of 26 letters—it's an ordered list of characters. Each letter occurs in a specific position within the list. "Z" is more than merely a letter of the alphabet—specifically, "Z" is the alphabet's 26th letter. Also, recall that lowercase letters have higher numeric values than uppercase letters, so "a" is greater than "A."

How do the string-comparison functions know that one particular letter comes before another? All characters are represented inside the computer as **numeric codes**

in character sets such as ASCII and Unicode; when the computer compares two strings, it actually compares the characters' numeric codes in each string. This is called a lexicographical comparison. See Appendix B for the ASCII characters' numeric values. ASCII is a subset of the Unicode character set.

The negative and positive values returned by strcmp and strncmp are *implementation-dependent*. For some, these values are -1 or 1, as in Fig. 8.13. For others, the values returned are the difference between the numeric codes of the first different characters in each string. For this program's comparisons, that's the difference between the numeric codes of "N" in "New" and "H" in "Holidays"—6 or -6, depending on which string is the first argument.

\checkmark

Self Check

- I (*Multiple Choice*) Which of the following statements about functions strcmp and strncmp is *false*?
 - a) Function strcmp compares its first string argument with its second string argument, character-by-character.
 - b) Function strcmp returns 0 if the strings are equal, a negative value if the first string is less than the second and a positive value if the first string is greater than the second.
 - c) Function strncmp is equivalent to strcmp but compares up to a specified number of characters.
- d) Function strncmp compares characters following a null character in a string. **Answer:** d) is *false*. Actually, function strncmp does not compare characters following a null character in a string.
- **2** (*Discussion*) How do the string-comparison functions strcmp and strncmp know that one particular letter "comes before" another?

Answer: All characters are represented inside the computer as numeric codes in character sets such as ASCII and Unicode. When the computer compares two strings, it compares the characters' numeric codes. This is called a lexicographical comparison.

8.8 Search Functions of the String-Handling Library

This section presents the string-handling library functions that search strings for characters and other strings, summarized in the following table.

Function prototypes and descriptions

```
char *strchr(const char *s, int c);
```

Locates the first occurrence of character c in string s. If c is found, strchr returns a pointer to c in s. Otherwise, a NULL pointer is returned.

```
size_t strcspn(const char *s1, const char *s2);
```

Determines and returns the length of the initial segment of string s1 consisting of characters not contained in string s2.

Function prototypes and descriptions

```
size_t strspn(const char *s1, const char *s2);
      Determines and returns the length of the initial segment of string $1 consisting only of
      characters contained in string s2.
char *strpbrk(const char *s1, const char *s2);
      Locates the first occurrence in string s1 of any character in string s2. If a character from s2
      is found, strpbrk returns a pointer to that character in s1. Otherwise, it returns NULL.
char *strrchr(const char *s, int c);
     Locates the last occurrence of c in string s. If c is found, strrchr returns a pointer to c in
     string s. Otherwise, it returns NULL.
char *strstr(const char *s1, const char *s2);
      Locates the first occurrence in string s1 of string s2. If the string is found, strstr returns a
      pointer to the string in $1. Otherwise, it returns NULL.
char *strtok(char *s1, const char *s2);
     A sequence of calls to strtok breaks string s1 into tokens separated by characters contained
     in string 52. Tokens are logical pieces, such as words in a line of text. The first call uses 51 as
      the first argument. Subsequent calls to continue tokenizing the same string require NULL as
      the first argument. Each call returns a pointer to the current token. If there are no more
      tokens, strtok returns NULL.
```

8.8.1 Function strchr

Function **strchr** searches for the first occurrence of a character in a string. If the character is found, strchr returns a pointer to the character in the string; otherwise, strchr returns NULL. Figure 8.14 searches for the first occurrences of 'a' and 'z' in "This is a test".

```
// fig08_14.c
2 // Using function strchr
3 #include <stdio.h>
4 #include <string.h>
5
6
   int main(void) {
7
       const char *string = "This is a test"; // initialize char pointer
8
       char character1 = 'a';
9
       char character2 = 'z';
10
       // if character1 was found in string
П
12
       if (strchr(string, character1) != NULL) { // can remove "!= NULL"
          printf("\'%c\' was found in \"%s\".\n", character1, string);
13
       }
14
15
       else { // if character1 was not found
          printf("\'%c\' was not found in \"%s\".\n", character1, string);
16
17
       }
```

Fig. 8.14 | Using function strchr. (Part 1 of 2.)

```
18
19
       // if character2 was found in string
       if (strchr(string, character2) != NULL) { // can remove "!= NULL"
20
          printf("\'%c\' was found in \"%s\".\n", character2, string);
21
22
23
       else { // if character2 was not found
          printf("\'%c\' was not found in \"%s\".\n", character2, string);
24
25
26 }
'a' was found in "This is a test".
'z' was not found in "This is a test".
```

Fig. 8.14 Using function strchr. (Part 2 of 2.)

8.8.2 Function strcspn

Function **strcspn** (Fig. 8.15) determines the length of the initial part of its first string argument that does *not* contain any characters from its second string argument. The function returns the segment's length.

```
// fig08_15.c
2 // Using function strcspn
3 #include <stdio.h>
4 #include <string.h>
5
6 int main(void) {
7
      // initialize two char pointers
       const char *string1 = "The value is 3.14159";
8
       const char *string2 = "1234567890";
9
10
       printf("string1 = %s\nstring2 = %s\n\n%s\n%s%zu\n", string1, string2,
11
12
          "The length of the initial segment of string1",
13
          "containing no characters from string2 = ",
          strcspn(string1, string2));
14
15 }
string1 = The value is 3.14159
string2 = 1234567890
The length of the initial segment of string1
containing no characters from string2 = 13
```

Fig. 8.15 | Using function strcspn.

8.8.3 Function strpbrk

Function **strpbrk** searches its first string argument for the *first occurrence* of any character in its second string argument. If a character from the second argument is found, strpbrk returns a pointer to the character in the first argument; otherwise, it returns NULL. Figure 8.16 locates the first occurrence in string1 of any character from string2.

```
// fig08_16.c
2 // Using function strpbrk
3 #include <stdio.h>
4 #include <string.h>
5
6 int main(void) {
7
       const char *string1 = "This is a test";
       const char *string2 = "beware";
8
9
       printf("%s\"%s\"\n'%c'%s \"%s\"\n",
10
          "Of the characters in ", string2, *strpbrk(string1, string2),
11
          " appears earliest in ", string1);
12
13 }
Of the characters in "beware"
'a' appears earliest in "This is a test"
```

Fig. 8.16 | Using function strpbrk.

8.8.4 Function strrchr

Function **strrchr** searches for the last occurrence of the specified character in a string. If the character is found, **strrchr** returns a pointer to the character in the string; otherwise, it returns NULL. Figure 8.17 searches for the last occurrence of the character 'z' in the string "A zoo has many animals including zebras".

```
I // fig08_17.c
2 // Using function strrchr
3 #include <stdio.h>
4 #include <string.h>
5
6 int main(void) {
7
       const char *s1 = "A zoo has many animals including zebras";
8
       int c = 'z'; // character to search for
9
       printf("%s '%c' %s\n\"%s\"\n",
10
П
          "Remainder of s1 beginning with the last occurrence of character",
12
         c, "is:", strrchr(s1, c));
13
Remainder of s1 beginning with the last occurrence of character 'z' is:
"zebras"
```

Fig. 8.17 Using function strrchr.

8.8.5 Function strspn

Function **strspn** (Fig. 8.18) determines the length of the initial part of its first argument containing only characters from the string in its second argument. The function returns the length of the segment.

```
// fig08_18.c
2 // Using function strspn
3 #include <stdio.h>
4 #include <string.h>
5
6
  int main(void) {
7
       const char *string1 = "The value is 3.14159";
       const char *string2 = "aehi lsTuv";
9
10
       printf("string1 = %s\nstring2 = %s\n\n%s\n%s%zu\n", string1, string2,
          "The length of the initial segment of string1",
11
          "containing only characters from string2 = ",
12
          strspn(string1, string2));
13
14 }
string1 = The value is 3.14159
string2 = aehi 1sTuv
The length of the initial segment of string1
containing only characters from string2 = 13
```

Fig. 8.18 | Using function strspn.

8.8.6 Function strstr

Function **strstr** searches for the first occurrence of its second string argument in its first string argument. If the second string is found in the first, **strstr** returns a pointer to the second string's location in the first. Figure 8.19 uses **strstr** to find the string "def" in the string "abcdefabcdef".

```
I // fig08_19.c
2 // Using function strstr
3 #include <stdio.h>
4 #include <string.h>
5
6 int main(void) {
       const char *string1 = "abcdefabcdef";
7
8
       const char *string2 = "def"; // string to search for
9
10
       printf("string1 = %s\nstring2 = %s\n\n%s\n%s%s\n", string1, string2,
11
          "The remainder of string1 beginning with the",
          "first occurrence of string2 is: ", strstr(string1, string2));
12
13 }
string1 = abcdefabcdef
string2 = def
The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef
```

Fig. 8.19 | Using function strstr.

8.8.7 Function strtok

Function **strtok** (Fig. 8.20) breaks a string into a series of **tokens**—also called tokenizing the string. A token is a sequence of characters separated by delimiters, such as *spaces* or *punctuation marks*. A delimiter can be any character. For example, in a line of text, each word is a token, and the spaces and punctuation separating the words are delimiters. You can change the delimiter string in each strtok call. Figure 8.20 tokenizes the string "This is a sentence with 7 tokens" and prints the tokens. Function strtok mod*ifies the input string* by placing '\0' at the end of each token, so copy the string if you intend to use it after the calls to strtok. See CERT recommendation STR06-C for the SEC problems with assuming that strtok does not modify the string in its first argument

```
// fig08_20.c
2
   // Using function strtok
3 #include <stdio.h>
  #include <string.h>
5
6
   int main(void) {
7
       char string[] = "This is a sentence with 7 tokens";
8
9
       printf("The string to be tokenized is:\n%s\n\n", string);
10
       puts("The tokens are:");
11
12
       char *tokenPtr = strtok(string, " "); // begin tokenizing sentence
13
14
       // continue tokenizing sentence until tokenPtr becomes NULL
15
       while (tokenPtr != NULL) {
          printf("%s\n", tokenPtr);
16
          tokenPtr = strtok(NULL, " "); // get next token
17
18
       }
19
    }
The string to be tokenized is:
This is a sentence with 7 tokens
The tokens are:
This
is
sentence
with
tokens
```

Using function strtok. Fig. 8.20

First strtok Call

Multiple calls to strtok are required to tokenize a string, assuming it contains more than one token. The first call to strtok (line 12) receives as arguments a string to tokenize and a string containing characters that separate the tokens. The statement

```
char *tokenPtr = strtok(string, " "); // begin tokenizing sentence
```

assigns tokenPtr a pointer to the first token in string. The second argument, " ", indicates that tokens are separated by spaces. Function strtok searches for the first character in string that's not a delimiter (space). This begins the first token. The function then finds the next delimiter in the string and *replaces it with a null ('\0')* character to terminate the current token. Function strtok saves a pointer to the character following that token in string and returns a pointer to the current token.

Subsequent strtok Calls

Subsequent strtok calls in line 17 continue tokenizing string. These calls receive *NULL as their first argument* to indicate that they should continue tokenizing from the location in string saved by the last call. If no tokens remain, strtok returns NULL.

Self Check

- **I** (*Multiple Choice*) Which function is described by "Locates the first occurrence in string s1 of string s2—if the string is found, the function returns a pointer to the string in s1; otherwise, it returns NULL"?
 - a) strpbrk.
 - b) strstr.
 - c) strspn.
 - d) strcspn.

Answer: b. strstr.

2 (*Fill-In*) In the context of function strtok, a _____ is a sequence of characters separated by delimiters.

Answer: token.

8.9 Memory Functions of the String-Handling Library

The string-handling library functions in this section manipulate, compare and search blocks of memory. These functions treat memory as character arrays and can manipulate any block of data. The following table summarizes the string-handling library's memory functions. In the function discussions, "object" refers to a block of data.

Function prototype **Function description** int memcmp(const void *s1, const void *s2, size_t n); Compares the first n bytes of the objects pointed to by s1 and s2. The function returns 0, less than 0 or greater than 0 if s1 is equal to, less than or greater than s2. void *memchr(const void *s, int c, size_t n); Locates the first occurrence of c (converted to unsigned char) in the first n bytes of the object pointed to by s. If c is found, memchr returns a pointer to c in the object; otherwise, it returns NULL. void *memset(void *s, int c, size_t n); Copies c (converted to unsigned char) into the first n bytes of the object pointed to by s, then returns a pointer to the result.

The pointer parameters are declared void *, so they can be used to manipulate memory for any data type. Recall from Chapter 7 that any pointer can be assigned directly to a void * pointer, and a void * pointer can be assigned directly to a pointer of any other type. Because a void * pointer cannot be dereferenced, each function receives a size argument that specifies the number of bytes the function will process. For simplicity, the examples in this section manipulate character arrays (blocks of characters). The preceding table's functions *do not* check for terminating null characters because they manipulate blocks of memory that are not necessarily strings.

8.9.1 Function memcpy

Function memcpy copies a specified number of bytes from the object pointed to by its second argument into the one pointed to by its first argument. The function can receive a pointer to any type of object. Its result is *undefined* if the two objects overlap in memory—that is, they're parts of the same object. In such cases, use memmove instead. Figure 8.21 uses memcpy to copy the string in array s2 to array s1. Function memcpy is Representation of PERF more efficient than strcpy when you know the size of the string you're copying.

```
I // fig08_21.c
 2 // Using function memcpy
 3 #include <stdio.h>
 4 #include <string.h>
 5
 6
   int main(void) {
        char s1[17] = "";
 7
        char s2[] = "Copy this string";
 8
 9
10
        \frac{\mathsf{memcpy}(\mathsf{s1, s2, 17)}}{\mathsf{memcpy}(\mathsf{s1, s2, 17)}} // 17 so we copy \mathsf{s2's} terminating \0
11
        puts("After s2 is copied into s1 with memcpy, s1 contains:");
12
        puts(s1);
    }
13
```

Using function memcpy. (Part 1 of 2.)

```
After s2 is copied into s1 with memcpy, s1 contains:
Copy this string
```

Fig. 8.21 | Using function memcpy. (Part 2 of 2.)

8.9.2 Function memmove

Like memcpy, function memmove copies a specified number of bytes from the object pointed to by its second argument into the object pointed to by its first argument. Copying is performed as if the bytes were copied from the second argument into a temporary array, then copied from the temporary array into the first argument. This allows bytes from one part of a string (or block of memory) to be copied into another part of the *same* string (or block of memory), even if the two portions overlap. Other than memmove, string-manipulation functions that copy characters have undefined results when copying between parts of the same string. Figure 8.22 uses memmove to copy the last 10 bytes of array x into the first 10 bytes of array x.

```
| // fig08_22.c
2 // Using function memmove
3 #include <stdio.h>
4 #include <string.h>
5
6
  int main(void) {
7
       char x[] = "Home Sweet Home"; // initialize char array x
8
9
       printf("The string in array x before memmove is: %s\n", x);
       printf("The string in array x after memmove is: %s\n",
10
          (char *) memmove(x, &x[5], 10));
П
12
   }
```

```
The string in array x before memmove is: Home Sweet Home
The string in array x after memmove is: Sweet Home Home
```

Fig. 8.22 Using function memmove.

8.9.3 Function memcmp

Function memcmp (Fig. 8.23) compares the specified number of bytes of its first argument with its second argument's corresponding bytes. The function returns a value greater than 0 if the first argument is greater than the second, 0 if the arguments are equal or a value less than 0 if the first argument is less than the second.

```
1 // fig08_23.c
2 // Using function memcmp
3 #include <stdio.h>
4 #include <string.h>
```

Fig. 8.23 | Using function memcmp. (Part 1 of 2.)

```
5
    int main(void) {
7
         char s1[] = "ABCDEFG";
 8
         char s2[] = "ABCDXYZ";
 9
         printf("s1 = %s\ns2 = %s\n\n%s%2d\n%s%2d\n", s1, s2,
10
            "memcmp(s1, s2, 4) = ", memcmp(s1, s2, 4),
"memcmp(s1, s2, 7) = ", memcmp(s1, s2, 7),
"memcmp(s2, s1, 7) = ", memcmp(s2, s1, 7));
П
12
13
14
   }
s1 = ABCDEFG
s2 = ABCDXYZ
memcmp(s1, s2, 4) = 0
memcmp(s1, s2, 7) = -1
memcmp(s2, s1, 7) = 1
```

Fig. 8.23 Using function memcmp. (Part 2 of 2.)

8.9.4 Function memchr

Function memchr searches for the first occurrence of a byte, represented as unsigned char, in the specified number of bytes of an object. If the byte is found, memchr returns a pointer to the byte in the object; otherwise, it returns NULL. Figure 8.24 searches for the byte containing 'r' in the string "This is a string".

```
| // fig08_24.c
2 // Using function memchr
3 #include <stdio.h>
4 #include <string.h>
5
6
  int main(void) {
7
      const char *s = "This is a string";
8
       printf("The remainder of s after character 'r' is found is \"%s\"\n",
9
          (char *) memchr(s, 'r', 16));
10
11
  }
The remainder of s after character 'r' is found is "ring"
```

Fig. 8.24 | Using function memchr.

8.9.5 Function memset

Function memset copies the value of the byte in its second argument into the first nbytes of the object pointed to by its first argument, where n is specified by the third argument. You can use memset to set an array's elements to 0 rather than assigning 0 = PERF to each element. For example, a five-element int array n could be reset to 0s with

```
memset(n, 0, 5);
```

Many hardware architectures have a block copy or clear instruction that the compiler can use to optimize memset for high-performance zeroing of memory. Figure 8.25 uses memset to copy 'b' into the first 7 bytes of string1.

```
// fig08_25.c
1
2
   // Using function memset
3 #include <stdio.h>
  #include <string.h>
5
   int main(void) {
7
      char string1[15] = "BBBBBBBBBBBBBBBB";
8
      printf("string1 = %s\n", string1);
9
      printf("string1 after memset = %s\n", (char *) memset(string1, 'b', 7));
10
11
string1 after memset = bbbbbbbBBBBBBBB
```

Fig. 8.25 | Using function memset.

Self Check

- **I** (*Multiple Choice*) Which of the following statements about function memcpy is *false*?
 - a) The function copies a specified number of bytes from the object pointed to by its second argument into the object pointed to by its first argument.
 - b) The function can receive a pointer to any type of object.
 - c) The result of this function is undefined if the two objects are completely separate in memory.
 - d) The function is more efficient than strcpy when you know the size of the string you're copying.

Answer: c) is *false*. Actually, the result is undefined if the two objects overlap in memory—that is, they're parts of the same object. In such cases, use memmove.

- **2** (*Fill-In*) The memory-handling functions of the string-handling library manipulate, compare and search blocks of memory, which the functions treat as _____. Answer: character arrays.
- **3** (*TruelFalse*) Function memmove copies a specified number of bytes from the object pointed to by its second argument into the object pointed to by its first argument. Copying is performed as if the bytes were copied from the second argument into a temporary array, then copied from the temporary array into the first argument. This allows bytes from one block of memory to be copied into another part of the *same* block of memory, even if the two portions overlap.

Answer: True.

8.10 Other Functions of the String-Handling Library

The two remaining string-handling library functions are strerror and strlen, which are summarized in the following table.

```
Function prototype Function description

char *strerror(int errornum);

Maps errornum to a full text string in a compiler- and locale-specific manner and returns the string. Error numbers are defined in errno.h.

size_t strlen(const char *s);

Returns the length of string s—that is, the number of characters preceding the string's terminating null character.
```

8.10.1 Function strerror

Function **strerror** takes an error number and creates an error message string. A pointer to the string is returned. Figure 8.26 demonstrates strerror.

```
// fig08_26.c
// Using function strerror
#include <stdio.h>
#include <string.h>

int main(void) {
   printf("%s\n", strerror(2));
}
No such file or directory
```

Fig. 8.26 | Using function strerror.

8.10.2 Function strlen

Function strlen takes a string as an argument and returns the number of characters in the string—the terminating null character is not included in the length. Figure 8.27 demonstrates function strlen.

```
// fig08_27.c
// Using function strlen
#include <stdio.h>
#include <string.h>

int main(void) {
    const char *string1 = "abcdefghijklmnopqrstuvwxyz";
    const char *string2 = "four";
    const char *string3 = "Boston";
```

Fig. 8.27 | Using function strlen. (Part I of 2.)

Fig. 8.27 | Using function strlen. (Part 2 of 2.)

Self Check

1 (*True/False*) The error message strings returned by function strerror are uniform across platforms.

Answer: False. The messages vary by compiler and locale.

2 (*True/False*) Function strlen takes a string as an argument and returns the number of characters in the string, including the terminating null character.

Answer: *False.* Actually, the terminating null character is not included in the length.

8.11 Secure C Programming

Secure String-Processing Functions

Earlier Secure C Programming sections covered C11's functions printf_s and scan-f_s. This chapter presented functions sprintf, strcpy, strncpy, strcat, strncat, strtok, strlen, memcpy, memmove and memset. The C11 standard's *optional* Annex K provides versions of these and many other string-processing and input/output functions. If your C compiler supports Annex K, consider using the secure versions of these functions. Among other things, the Annex K versions help prevent buffer overflows by requiring additional parameters that specify the number of elements in a target array and by ensuring that pointer arguments are non-NULL.

Reading Numeric Inputs and Input Validation

It's important to validate the data that you input into a program. For example, when you ask the user to enter an int in the range 1–100, then attempt to read that int using scanf, there are several possible problems. The user could enter:

- an int that's outside the program's required range (such as 102).
- an int that's outside that computer's allowed range for ints (such as 8,000,000,000 on a machine with 32-bit ints).
- a non-integer numeric value (such as 27.43).
- a non-numeric value (such as FOVR).

You can use various functions that you learned in this chapter to fully validate such input. For example, you could

- use fgets to read the input as a line of text,
- convert the string to a number using strtol and ensure that the conversion was successful, then
- ensure that the value is in range.

For more information and techniques for converting input to numeric values, see CERT guideline INT05-C at https://wiki.sei.cmu.edu/.

Self Check

- **I** (*Fill-In*) Among other things, the secure string-processing functions of Annex K help prevent buffer ______ by requiring additional parameters that specify the number of elements in a target array and by ensuring that pointer arguments are non-NULL. **Answer:** overflows.
- **2** (*True/False*) It's important to validate the data you input into a program. You can use various string-and-character-processing functions to fully validate inputs. For example, you could use fgets to read the input as a line of text, convert the string to a number using strto1, ensure the conversion was successful, then ensure that the value is in range.

Answer: True.

Summary

Section 8.2 Fundamentals of Strings and Characters

- Characters are the fundamental building blocks of source programs. Every program is composed of a sequence of characters that—when grouped together meaningfully—is interpreted by the computer as instructions used to accomplish a task.
- A character constant (p. 388) is an int value represented as a character in single quotes. The value of a character constant is the character's integer value in the machine's character set (p. 389).
- A string (p. 389) is a series of characters treated as a single unit. A string may include letters, digits and various special characters (p. 389) such as +, -, *, / and \$. String literals, or string constants, are written in double quotation marks.
- A string in C is an array of characters ending in the null character (p. 389; '\0').
- A string is accessed via a **pointer** to its first character. The value of a string is the **address** of its first character.
- A character array or a variable of type char * can be initialized with a string in a definition.
- When defining a character array to contain a string, the array must be large enough to store
 the string and its terminating null character.
- A string can be stored in an array using scanf. Function scanf will read characters until a space, tab, newline or end-of-file indicator is encountered.
- For a character array to be printed as a string, the array must contain a terminating null character.

Section 8.3 Character-Handling Library

- Function **isdigit** (p. 391) determines whether its argument is a **digit** (0-9).
- Function **isalpha** (p. 391) determines whether its argument is an **uppercase letter** (A–Z) or a **lowercase letter** (a–z).
- Function isalnum (p. 391) determines whether its argument is an uppercase letter (A–Z), a lowercase letter (a–z) or a digit (0–9).
- Function **isxdigit** (p. 391) determines whether its argument is a **hexadecimal digit** (p. 391; A-F, a-f, 0-9).
- Function islower (p. 393) determines whether its argument is a lowercase letter (a-z).
- Function isupper (p. 393) determines whether its argument is an uppercase letter (A–Z).
- Function toupper (p. 393) converts a lowercase letter to uppercase and returns it.
- Function tolower (p. 393) converts an uppercase letter to lowercase and returns it.
- Function isspace (p. 394) determines whether its argument is one of the following whitespace characters: ' '(space), '\f', '\n', '\r', '\t' or '\v'.
- Function **iscntr1** (p. 394) determines whether its argument is one of the following **control characters**: '\t', '\v', '\f', '\a', '\b', '\r' or '\n'.
- Function **ispunct** (p. 394) determines whether its argument is a **printing character** other than a space, a digit or a letter.
- Function **isprint** (p. 394) determines whether its argument is any printing character, including the space character.
- Function **isgraph** (p. 394) determines whether its argument is a printing character other than the space character.

Section 8.4 String-Conversion Functions

- Function **strtod** (p. 396) converts a sequence of characters representing a floating-point value to double. The location specified by its pointer to char * argument is assigned the remainder of the string after the conversion, or to the entire string if no portion of the string can be converted.
- Function **strto1** (p. 397) converts a sequence of characters representing an integer to long. This function works identically to strtod, but the third argument specifies the base of the value being converted.
- Function **strtoul** (p. 398) works identically to strtol but converts a sequence of characters representing an integer to unsigned long int.

Section 8.5 Standard Input/Output Library Functions

- Function **fgets** (p. 399) reads characters until a newline character or the end-of-file indicator is encountered. The arguments to fgets are an array of type char, the maximum number of characters to read and the stream from which to read. A null character ('\0') is appended to the array after reading terminates. If a newline is encountered, it's included in the input string.
- Function **putchar** (p. 399) prints its character argument.
- Function **getchar** (p. 401) reads a single character from the standard input and returns it as an integer. If the end-of-file indicator is encountered, getchar returns EOF.
- Function **puts** (p. 401) takes a string (char *) as an argument and prints the string followed by a newline character.
- Function **sprintf** (p. 401) uses the same conversion specifications as function printf to print formatted data into an array of type char.

• Function **sscanf** (p. 402) uses the same conversion specifications as function scanf to read formatted data from a string.

Section 8.6 String-Manipulation Functions of the String-Handling Library

- Function **strcpy** copies its second argument string into its first argument char array. You must ensure that the array is large enough to store the string and its terminating null character.
- Function **strncpy** (p. 404) is equivalent to strcpy, but specifies the maximum number of characters to copy from the string into the array. The terminating null character will be copied only if the number of characters to be copied is one more than the string's length.
- Function **strcat** (p. 405) appends its second argument string—including its terminating null character—to its first argument string. The first character of the second string replaces the null ('\0') character of the first string. You must ensure that the array used to store the first string is large enough to store both the first string and the second string.
- Function **strncat** (p. 404) appends a specified number of characters from the second string to the first string. A terminating null character is appended to the result.

Section 8.7 Comparison Functions of the String-Handling Library

- Function **strcmp** (p. 406) compares its first string argument to its second string argument, character by character. It returns 0 if the strings are equal, a negative value if the first string is less than the second or a positive value if the first string is greater than the second.
- Function **strncmp** (p. 406) is equivalent to strcmp, except that strncmp compares a specified number of characters. If one of the strings is shorter than the number of characters specified, strncmp compares characters until the null character in the shorter string is encountered.

Section 8.8 Search Functions of the String-Handling Library

- Function **strchr** (p. 409) searches for the first occurrence of a character in a string. If found, strchr returns a pointer to the character in the string; otherwise, strchr returns NULL.
- Function **strcspn** (p. 410) determines the length of the initial part of the string in its first argument that does not contain any characters from the string in its second argument. The function returns the segment's length.
- Function **strpbrk** (p. 410) searches for the first occurrence in its first argument of any character in its second argument. If a character from the second argument is found, **strpbrk** returns a pointer to the character; otherwise, **strpbrk** returns NULL.
- Function **strrchr** (p. 411) searches for the last occurrence of a character in a string. If found, **strrchr** returns a pointer to the character in the string; otherwise, **strrchr** returns NULL.
- Function **strspn** (p. 412) determines the length of the initial part of the string in its first argument that contains only characters from the string in its second argument. The function returns the length of the segment.
- Function **strstr** (p. 412) searches for the first occurrence of its second string argument in its first string argument. If the second string is found in the first string, a pointer to the location of the string in the first argument is returned.
- A sequence of calls to **strtok** (p. 413) breaks its first string argument into tokens (p. 413) that are separated by characters contained in the second string argument. The first call contains the string to tokenize as the first argument. Subsequent calls to continue tokenizing that string contain NULL as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, it returns NULL.

Section 8.9 Memory Functions of the String-Handling Library

- Function **memcpy** (p. 415) copies a specified number of bytes from the object to which its second argument points into the object to which its first argument points. The function can receive a pointer to any type of object.
- Function memmove (p. 416) copies a specified number of bytes from the object pointed to by its second argument to the object pointed to by its first argument. Copying is accomplished as if the bytes were copied from the second argument to a temporary array and then copied from the temporary array to the first argument.
- Function memcmp (p. 416) compares the specified number of bytes of its first and second arguments.
- Function memchr (p. 417) searches for the first occurrence of a byte, represented as unsigned char, in the specified number of bytes of an object. If the byte is found, a pointer to the byte is returned; otherwise, a NULL pointer is returned.
- Function memset (p. 417) copies its second argument, treated as an unsigned char, to a specified number of bytes of the object pointed to by the first argument.

Section 8.10 Other Functions of the String-Handling Library

- Function **strerror** (p. 419) maps an integer error number into a full text string in a locale-specific manner. A pointer to the string is returned.
- Function **strlen** (p. 419) takes a string as an argument and returns the **number of characters** in the string—the terminating null character is not included in the length of the string.

Self-Review Exercises

- **8.1** Write a single statement to accomplish each of the following. Assume variable c is a char, variables x, y and z are ints, variables d, e and f are doubles, variable ptr is a char * and s1 and s2 are 100-element char arrays.
 - a) Convert the character stored in variable c to an uppercase letter. Assign the result to variable c.
 - b) Determine whether the value of variable c is a digit. Use the conditional operator as shown in Figs. 8.1–8.3 to print " is a " or " is not a " when the result is displayed.
 - c) Determine whether the value of variable c is a control character. Use the conditional operator to print " is a " or " is not a " when the result is displayed.
 - d) Read a line of text into array s1 from the keyboard. Do not use scanf.
 - e) Print the line of text stored in array \$1. Do not use printf.
 - f) Assign ptr the location of the last occurrence of c in s1.
 - g) Print the value of variable c. Do not use printf.
 - h) Determine whether the value of c is a letter. Use the conditional operator to print " is a " or " is not a " when the result is displayed.
 - i) Read a character from the keyboard and store the character in variable c.
 - j) Assign ptr the location of the first occurrence of s2 in s1.
 - k) Determine whether the value of variable c is a printing character. Use the conditional operator to print " is a " or " is not a " when the result is displayed.

- l) Read three double values into variables d, e and f from the string "1.27 10.3 9.432".
- m) Copy the string stored in array s2 into array s1.
- n) Assign ptr the location of the first occurrence in s1 of any character from s2.
- o) Compare the string in s1 with the string in s2. Print the result.
- p) Assign ptr the location of the first occurrence of c in s1.
- q) Use sprintf to print the values of integer variables x, y and z into array s1. Each value should be printed with a field width of 7.
- r) Append 10 characters from the string in s2 to the string in s1.
- s) Determine the length of the string in s1. Print the result.
- t) Assign ptr to the location of the first token in s2. Tokens in the string s2 are separated by commas (,).
- 8.2 Show two different ways to initialize char array vowel with the string "AEIOU".
- **8.3** What, if anything, prints when each of the following C statements is performed? If the statement contains an error, describe the error and indicate how to correct it. Assume the following variable definitions:

```
char s1[50] = "jack";
char s2[50] = "jill";
char s3[50] = "";

a) printf("%c%s", toupper(s1[0]), &s1[1]);
b) printf("%s", strcpy(s3, s2));
c) printf("%s", strcat(strcat(strcpy(s3, s1), " and "), s2));
d) printf("%zu", strlen(s1) + strlen(s2));
e) printf("%zu", strlen(s3)); // using s3 after part (c) executes
```

8.4 Find the error in each of the following and explain how to correct it:

```
a) char s[10] = "";
    strncpy(s, "hello", 5);
    printf("%s\n", s);
b) printf("%s", 'a');
c) char s[12] = "";
    strcpy(s, "Welcome Home");
d) if (strcmp(string1, string2)) {
        puts("The strings are equal");
    }
```

Answers to Self-Review Exercises

8.1 See the answers below:

```
e) puts(s1);
     f) ptr = strrchr(s1, c);
     g) putchar(c);
     h) printf("'\%c'\%sletter\n", c, isalpha(c) ? " is a " : " is not a ");
     i) c = getchar();
     j) ptr = strstr(s1, s2);
     k) printf("'%c'%sprinting character\n",
           c, isprint(c) ? " is a " : " is not a ");
     l) sscanf("1.27 10.3 9.432", "%f%f%f", &d, &e, &f);
     m)strcpy(s1, s2);
     n) ptr = strpbrk(s1, s2);
     o) printf("strcmp(s1, s2) = %d\n", strcmp(s1, s2));
     p) ptr = strchr(s1, c);
     q) sprintf(s1, "%7d%7d%7d", x, y, z);
     r) strncat(s1, s2, 10);
     s) printf("strlen(s1) = %zu\n", strlen(s1));
     t) ptr = strtok(s2, ",");
8.2 char vowel[] = "AEIOU";
     char vowel[] = {'A', 'E', 'I', 'O', 'U', '\0'};
8.3
     See the answers below:
     a) Jack
     b) jill
     c) jack and jill
     d) 8
     e) 13
```

- **8.4** See the answers below:
 - a) Error: Function strncpy does not write a terminating null character to array s, because its third argument is equal to the length of the string "hello". Correction: Make the third argument of strncpy 6, or assign '\0' to s[5].
 - b) Error: Attempting to print a character constant as a string.

 Correction: Use %c to output the character, or replace 'a' with "a".
 - c) Error: Character array s is not large enough to store the terminating null character.
 - Correction: Declare the array with more elements.
 - d) Error: Function strcmp returns 0 if the strings are equal; therefore, the condition in the if statement is false, and the printf will not execute.

 Correction: Compare the result of strcmp with 0 in the condition.

Exercises

8.5 *(Character Testing)* Write a program that inputs a character from the keyboard and tests it with each of the character-handling library functions. The program should print the value returned by each function.

- **8.6** (*Displaying Strings in Uppercase and Lowercase*) Write a program that inputs a line of text into char array s[100]. Display the line in uppercase letters and in lowercase letters.
- **8.7** (Converting Strings to Integers for Calculations) Write a program that inputs four strings representing integers, converts the strings to integers, sums the values and prints the total of the four values.
- **8.8** (Converting Strings to Floating Point for Calculations) Write a program that inputs four strings representing floating-point values, converts the strings to double values, sums the values and prints the total of the four values.
- **8.9** (Comparing Strings) Write a program that uses function strcmp to compare two strings input by the user. The program should state whether the first string is less than, equal to or greater than the second string.
- **8.10** (Comparing Portions of Strings) Write a program that uses function strncmp to compare two strings input by the user. The program should input the number of characters to be compared, then display whether those characters from the first string are less than, equal to or greater than the second string.
- **8.11** (Random Sentences) Use random-number generation to create sentences. Your program should use four arrays of pointers to char called article, noun, verb and preposition. Create a sentence by selecting a word at random from each array in the following order: article, noun, verb, preposition, article and noun. The arrays should be filled as follows: The article array should contain the articles "the", "a", "one", "some" and "any"; the noun array should contain the nouns "boy", "girl", "dog", "town" and "car"; the verb array should contain the verbs "drove", "jumped", "ran", "walked" and "skipped"; the preposition array should contain the prepositions "to", "from", "over", "under" and "on".

As each word is picked, concatenate it to the previous words in an array large enough to hold the entire sentence. Separate the words by spaces. The final sentence should start with a capital letter and end with a period. Generate 20 such sentences. Modify your program to produce a short story consisting of several of these sentences. (How about the possibility of a random term-paper writer?)

- **8.12** (*Limericks*) A limerick is a humorous five-line verse in which the first and second lines rhyme with the fifth, and the third line rhymes with the fourth. Using techniques similar to those developed in Exercise 8.11, write a program that produces random limericks. Polishing this program to produce good limericks is a challenging problem, but the result will be worth the effort!
- **8.13** (*Pig Latin*) Write a program that encodes English-language phrases into pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form pig-Latin phrases. For simplicity, use the following algorithm:

To form a pig-Latin phrase from an English-language phrase, tokenize the phrase into words with function strtok. To translate each English word into a pig-Latin

word, place the first letter of the English word at the end of the English word and add the letters "ay". Thus the word "jump" becomes "umpjay", the word "the" becomes "hetay" and the word "computer" becomes "omputercay". Blanks between words remain as blanks. Assume the following: The English phrase consists of words separated by blanks, there are no punctuation marks, and all words have two or more letters. Function printLatinWord should display each word. [Hint: Each time strtok finds a token, pass the token pointer to function printLatinWord, and print the pig-Latin word. We've provided simplified pig-Latin conversion rules here. For more detailed rules and variations, visit https://en.wikipedia.org/wiki/Pig_latin.]

- **8.14** (*Tokenizing Telephone Numbers*) Write a program that inputs a telephone number as a string in the form (555) 555-5555. Use function strtok to extract as tokens the area code, the first three digits of the phone number and the last four digits of the phone number. Concatenate the phone number's seven digits into one string. Convert the area-code string and phone-number string to integers, then display both.
- **8.15** (*Displaying a Sentence with Its Words Reversed*) Write a program that inputs a line of text, tokenizes the line with function strtok and outputs the tokens in reverse order.
- **8.16** (Searching for Substrings) Write a program that inputs a line of text and a search string from the keyboard. Using function strstr, locate the first occurrence of the search string in the line of text. Assign the location to variable searchPtr of type char *. If the search string is found, print the remainder of the line of text beginning with the search string. Then, use strstr again to locate the next occurrence of the search string in the line of text. If a second occurrence is found, print the remainder of the line of text beginning with the second occurrence. [Hint: The second call to strstr should contain searchPtr + 1 as its first argument.]
- **8.17** (Counting the Occurrences of a Substring) Write a program based on Exercise 8.16 that inputs several lines of text and a search string and uses function strstr to determine the total occurrences of the search string in the lines of text. Print the result.
- **8.18** (Counting the Occurrences of a Character) Write a program that inputs several lines of text and a search character and uses function strchr to determine the total occurrences of the character in the lines of text.
- **8.19** (Counting the Letters of the Alphabet in a String) Write a program based on the program of Exercise 8.18 that inputs several lines of text and uses function strchr to determine the total occurrences of each letter of the alphabet in the lines of text. Uppercase and lowercase letters should be counted together. Store the totals for each letter in an array and print the values in tabular format after the totals have been determined.
- **8.20** (Counting the Number of Words in a String) Write a program that inputs several lines of text and uses strtok to count the total number of words. Assume that the words are separated by either spaces or newline characters.

- **8.21** (Alphabetizing a List of Strings) Use the string-comparison functions and the techniques for sorting arrays to write a program that alphabetizes a list of strings. Use the names of 10 or 15 towns in your area as data for your program.
- **8.22** Appendix B shows the numeric code representations for the ASCII character set. Study Appendix B, then state whether each of the following is *true* or *false*.
 - a) The letter "A" comes before the letter "B".
 - b) The digit "9" comes before the digit "0".
 - c) The commonly used symbols for addition, subtraction, multiplication and division all come before any of the digits.
 - d) The digits come before the letters.
 - e) If a sort program sorts strings into ascending sequence, then the program will place the symbol for a right parenthesis before the symbol for a left parenthesis.
- **8.23** (Strings Starting with "b") Write a program that reads a series of strings and prints only those beginning with the letter "b".
- **8.24** (Strings Ending with "ed") Write a program that reads a series of strings and prints only those that end with the letters "ed".
- **8.25** (*Printing Letters for Various ASCII Codes*) Write a program that inputs an ASCII code and prints the corresponding character.
- **8.26** (Write Your Own Character-Handling Functions) Using the ASCII character chart in Appendix B as a guide, write your own versions of the character-handling functions in Section 8.3.
- **8.27** (Write Your String-Conversion Functions) Write your own versions of the functions in Section 8.4 for converting strings to numbers.
- **8.28** (Write Your Own String-Copy and String-Concatenation Functions) Write two versions of each string-copy and string-concatenation function in Section 8.6. The first version should use array indexing, and the second should use pointers and pointer arithmetic.
- **8.29** (Write Your Own String-Comparison Functions) Write two versions of each string-comparison function in Fig. 8.13. The first version should use array indexing, and the second should use pointers and pointer arithmetic.
- **8.30** (Write Your Own String-Length Function) Write two versions of function strlen in Fig. 8.27. The first version should use array indexing, and the second should use pointers and pointer arithmetic.

Special Section: Advanced String-Manipulation Exercises

The preceding exercises are keyed to the text and designed to test the reader's understanding of fundamental string-manipulation concepts. This section contains intermediate and advanced problems that you should find challenging yet enjoyable. They vary considerably in difficulty. Some require an hour or two of programming.

Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects.

- **8.31** (*Text Analysis*) String-manipulation capabilities enable some rather interesting approaches to analyzing the writings of great authors. Much attention has been focused on whether William Shakespeare ever lived. Some scholars find substantial evidence that Christopher Marlowe actually penned the masterpieces attributed to Shakespeare. Researchers have used computers to find similarities in the writings of these two authors. This exercise examines three methods for analyzing texts with a computer.
 - a) Write a program that reads several lines of text and prints a table indicating the number of occurrences of each letter of the alphabet in the text. For example, the following phrase contains one "a," two "b's," no "c's," and so on:

To be, or not to be: that is the question:

b) Write a program that reads several lines of text and prints a table indicating the number of one-letter words, two-letter words, three-letter words, and so on, appearing in the text. For example, the phrase

Whether 'tis nobler in the mind to suffer contains

Word length	Occurrences
1	0
2	2
3	1
4	2 (including 'tis)
5	0
6	2
7	1

c) Write a program that reads several lines of text and prints a table indicating the number of occurrences of each different word in the text. The program should include the words in the table in the same order in which they appear in the text. For example, the lines

To be, or not to be: that is the question: Whether 'tis nobler in the mind to suffer

contain the words "to" three times, "be" two times, "or" once, and so on.

8.32 (*Printing Dates in Various Formats*) Dates are commonly printed in several different formats in business correspondence. Two of the more common formats are

07/21/2003 and July 21, 2003

Write a program that reads a date in the first format and prints it in the second format.

8.33 (Check Protection) Computers are frequently used in check-writing systems, such as payroll and accounts-payable applications. Many stories circulate regarding weekly paychecks being printed (by mistake) for amounts in excess of \$1 million. Weird amounts are printed by computerized check-writing systems because of human error and/or machine failure. Systems designers, of course, make every effort to build controls into their systems to prevent erroneous checks from being issued.

Another serious problem is someone intentionally altering a check amount then cashing it fraudulently. To prevent a dollar amount from being altered, most computerized check-writing systems employ a technique called *check protection*.

Checks designed for imprinting by computer contain a fixed number of spaces in which the computer may print an amount. Suppose a paycheck contains nine blank spaces in which the computer is supposed to print the amount of a weekly paycheck. If the amount is large, then all nine of those spaces will be filled—for example:

```
11,230.60 (check amount)
-----
123456789 (position numbers)
```

On the other hand, if the amount is less than \$1,000, then several of the spaces will ordinarily be left blank—for example,

```
99.87
-----
123456789
```

contains four blank spaces. If a check is printed with blank spaces, it's easier for someone to alter the amount of the check. To prevent such alteration, many checkwriting systems insert *leading asterisks* to protect the amount as follows:

```
****99.87
-----
123456789
```

Write a program that inputs a dollar amount to be printed on a check and then prints the amount in check-protected format with leading asterisks if necessary. Assume that nine spaces are available for printing an amount.

8.34 (Word Equivalent of a Check Amount) Continuing the discussion of the previous exercise: One common check-writing security method requires that the check amount be both written in numbers and "spelled out" in words. Even if someone is able to alter the numerical amount of the check, it's extremely difficult to change the amount in words. Write a program that inputs a numeric check amount and writes the word equivalent of the amount. For example, the amount 52.43 should be written as

```
FIFTY TWO and 43/100
```

8.35 (*Project: A Metric Conversion Program*) Write a program that assists the user with metric conversions. Allow the user to specify the names of the units as strings (i.e., centimeters, liters, grams, and so on for the metric system and inches, quarts, pounds, and so on for the English system) and should respond to simple questions such as

```
"How many inches are in 2 meters?"
"How many liters are in 10 quarts?"
```

Your program should recognize invalid conversions. For example, the following question is not meaningful—"feet" are length units while "kilograms" are mass units.

"How many feet are in 5 kilograms?"

8.36 (Cooking with Healthier Ingredients) Obesity in the United States is increasing at an alarming rate. Check the Centers for Disease Control and Prevention (CDC) webpage at www.cdc.gov/obesity/data/index.html, which contains United States obesity data and facts. As obesity increases, so do occurrences of related problems (e.g., heart disease, high blood pressure, high cholesterol, type 2 diabetes). Write a program that helps users choose healthier ingredients when cooking, and helps those allergic to certain foods (e.g., nuts, gluten) find substitutes. The program should read a recipe from the user and suggest healthier replacements for some of the ingredients. For simplicity, your program should assume the recipe has no abbreviations for measures such as teaspoons, cups, and tablespoons, and uses numerical digits for quantities (e.g., 1 egg, 2 cups) rather than spelling them out (one egg, two cups). Some common substitutions are shown in the following table. Your program should display a warning such as, "Always consult your physician before making significant changes to your diet."

Ingredient	Substitution
1 cup sour cream	1 cup yogurt
1 cup milk	1/2 cup evaporated milk and 1/2 cup water
1 teaspoon lemon juice	1/2 teaspoon vinegar
1 cup sugar	1/2 cup honey, 1 cup molasses or 1/4 cup agave nectar
1 cup butter	1 cup margarine or yogurt
1 cup flour	1 cup rye or rice flour
1 cup mayonnaise	1 cup cottage cheese or 1/8 cup mayonnaise and 7/8 cup yogurt
1 egg	2 tablespoons cornstarch, arrowroot flour or potato starch or 2 egg whites or 1/2 of a large banana (mashed)
1 cup milk	1 cup soy milk
1/4 cup oil	1/4 cup applesauce
white bread	whole-grain bread

Your program should take into consideration that replacements are not always one-for-one. For example, if a cake recipe calls for three eggs, it might reasonably use six egg whites instead. Conversion data for measurements and substitutes can be obtained at various websites. Your program should consider the user's health concerns, such as high cholesterol, high blood pressure, weight loss, gluten allergy, and so on. For high cholesterol, the program should suggest substitutes for eggs and dairy products; if the user wishes to lose weight, low-calorie substitutes for ingredients such as sugar should be suggested.

8.37 (*Spam Scanner*) Spam (or junk e-mail) costs U.S. organizations billions of dollars a year in spam-prevention software, equipment, network resources, bandwidth,

and lost productivity. Research online some of the most common spam e-mail messages and words, and check your own junk e-mail folder. Create a list of 30 words and phrases commonly found in spam messages. Write a program in which the user enters an e-mail message. Read the message into a large character array and ensure that the program does not attempt to insert characters past the end of the array. Then scan the message for each of the 30 keywords or phrases. For each occurrence of one of these within the message, add a point to the message's "spam score." Next, rate the likelihood that the message is spam, based on the number of points it received.

8.38 (SMS Language) Short Message Service (SMS) is a communications service that allows sending text messages of 160 or fewer characters between mobile phones. With the proliferation of mobile phone use worldwide, SMS is being used in many developing nations for political purposes (e.g., voicing opinions and opposition), reporting news about natural disasters, and so on. Because the length of SMS messages is limited, SMS Language—abbreviations of common words and phrases in mobile text messages, e-mails, instant messages, etc.—is often used. For example, "in my opinion" is "IMO" in SMS Language. Research SMS Language online. Write a program that lets the user enter a message using SMS Language, then translates it into English (or your own language). Also provide a mechanism to translate text written in English (or your own language) into SMS Language. One potential problem is that one SMS abbreviation could expand into a variety of phrases. For example, IMO (as used above) could also stand for "International Maritime Organization," "in memory of," etc.

8.39 (Gender Neutrality) In Exercise 1.6, you researched eliminating sexism in all forms of communication. You then described the algorithm you'd use to read through a paragraph of text and replace gender-specific words with gender-neutral equivalents. Create a program that reads a paragraph of text, then replaces gender-specific words with gender-neutral ones. Display the resulting gender-neutral text.

A Challenging String-Manipulation Project

8.40 (*Project: A Crossword-Puzzle Generator*) Most people have worked a crossword puzzle at one time or another, but few have ever attempted to generate one. Generating a crossword puzzle is a difficult problem. It's suggested here as a stringmanipulation project requiring substantial sophistication and effort. There are many issues you must resolve to get even the simplest crossword-puzzle generator program working. For example, how does one represent the grid of a crossword puzzle inside the computer? Should one use a series of strings, or perhaps two-dimensional arrays? You need a source of words (i.e., a computerized dictionary) that can be directly referenced by the program. In what form should these words be stored to facilitate the complex manipulations required by the program? The really ambitious reader will want to generate the "clues" portion of the puzzle in which the brief hints for each "across" word and each "down" word are printed for the puzzle worker. Merely printing a version of the blank puzzle itself is not a simple problem.

Pqyoaf X Nylfomigrob Qwbbfmh Mndogvk: Rboqlrut yua Boklnxhmywex

8.41 (*Pqyoaf X Nylfomigrob: Cuzqvbpcxo vlk Adzdujcjjl*) No doubt, you noticed the section title above and this exercise's title both look like gibberish. This is not a mistake! In this exercise, we continue our focus on security by introducing cryptography. You'll create functions that implement a Vigenère secret-key cipher.^{3,4} After encrypting and decrypting your own text, you can use your decrypt function with our secret key to decrypt the encrypted titles above.

Cryptography



Cryptography has been used for thousands of years^{5,6} and is critically important in today's connected world. Every day, cryptography is used behind the scenes to ensure that your Internet-based communications are private and secure. For example, most websites (including deitel.com) now use the HTTPS protocol to encrypt and decrypt your web interactions.

Caesar Cipher

Julius Caesar used a simple substitution cipher to encrypt military communications.⁷ Known as the Caesar cipher, his technique replaces every letter in a message with the letter three ahead in the alphabet. So, A is replaced with D, B with E, C with F, ... X with A, Y with B and Z with C. Thus, the unencrypted text

Caesar Cipher

would be encrypted as

Fdhvdu Flskhu

The encrypted text is known as ciphertext. The unencrypted text is known as plaintext or cleartext.

Experimenting with Ciphers

For a fun way to play with the Caesar cipher and many other cipher algorithms, visit:

https://cryptii.com/pipes/caesar-cipher

which is an online implementation of the open-source cryptii project:

https://github.com/cryptii/cryptii

On cryptii.com, you can enter plaintext, choose a cipher to use, specify that cipher's settings and view the resulting ciphertext.

- 3. "Crypto Corner—Vigenère Cipher." Accessed December 23, 2020. https://crypto.inter-active-maths.com/vigenegravere-cipher.html.
- 4. "Vigenère cipher." Accessed December 23, 2020. https://en.wikipedia.org/wiki/ Vigenère_cipher.
- 5. "Cryptography." Accessed December 23, 2020. https://en.wikipedia.org/wiki/Cryptography#History_of_cryptography_and_cryptanalysis.
- 6. Binance Academy, "History of Cryptography." Accessed December 23, 2020. https://www.binance.vision/security/history-of-cryptography.
- 7. "Caesar Cipher." Accessed December 23, 2020. https://en.wikipedia.org/wiki/Caesar_cipher.

Vigenère Cipher

Simple substitution ciphers like the Caesar cipher are relatively easy to decrypt. For example, "e" is the most frequently used English letter. So, you could study English ciphertext and assume that the most frequently appearing character probably is an "e."

The Vigenère secret-key cipher uses letters from the plaintext and a secret key to locate replacement characters in 26 Caesar ciphers—one for each letter of the alphabet. These 26 ciphers form a 26-by-26 two-dimensional array called the Vigenère square:

	Α	В	C	D	E	F	G	Н	I	J	K	L	M	N	0	P	Q	R	S	Т	U	٧	W	X	Υ	Z
Α	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	V	W	Χ	Υ	Ζ
В	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α
C	C	D	Ε	F	G	Н	Ι	J	K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В
D	D	Ε	F	G	Н	Ι	J	K	L	M	N	0	Р	Q	R	S	Т	U	V	W	Χ	Υ	Z	Α	В	C
Ε	Ε	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D
F	F	G	Н	Ι	J	K	L	М	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Е
G	G	Н	Ι	J	K	L	М	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F
Н	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	V	W	Χ	Υ	Z	Α	В	C	D	Ε	F	G
Ι	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н
J	J	K	L	М	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι
K	K	L	М	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J
L	L	M	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K
M	М	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L
N	Ν	0	Р	Q	R	S	Т	U	V	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M
0	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	N
P	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	N	0
Q	Q	R	S	Т	U	٧	W	Χ	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	Ν	0	Р
R	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	Ν	0	Р	Q
S	S	Т	U	٧	W	Χ	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R
Т	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	Ν	0	Р	Q	R	S
U	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	Ν	0	Р	Q	R	S	Т
V	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	Ν	0	Р	Q	R	S	Т	U
W	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	٧
X	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	N	0	Р	Q	R	S	Т	U	V	W
Y	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ
Z	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	N	0	Р	Q	R	S	Τ	U	٧	W	Χ	Υ

You look up substitutions using the bold blue letters that label the rows and columns.

Secret-Key Requirements

For the Vigenère cipher described here, the secret key must contain only letters. Like passwords, the secret key should not be easy to guess. To create the ciphertext in the titles at the beginning of this exercise, we used as our secret key the following 11 randomly selected characters:

XMWUJBVYHXZ

Your key can have as many characters as you like. The person decrypting the ciphertext must know the secret key used to create the ciphertext. Presumably, you'd provide that in advance—possibly in a face-to-face meeting. The secret key must, of course, be carefully guarded.

^{8.} There are many websites offering Vigenère cipher decoders that attempt to decrypt ciphertext without the original secret key. We tried several, but none restored our original text.

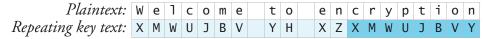
The Vigenère Cipher Encryption Algorithm

To see how the Vigenère cipher works, let's use the key "XMWUJBVYHXZ" and encrypt the plaintext string:

Welcome to encryption

Our encryption and decryption implementations preserve the plaintext's original case. Uppercase letters in the plaintext remain as uppercase in the ciphertext and vice versa, and lowercase letters in the plaintext remain as lowercase in the ciphertext and vice versa. We chose to pass non-letters in the plaintext—like spaces, digits and punctuation—through to the ciphertext and vice versa.

First, we repeat the secret key until the length matches the plaintext:



In the diagram above, we highlighted in light blue the secret key, then in darker blue the secret key's eight repeated letters.

We begin the encryption by using the first letter in the repeating key text ('X') to select a row in the Vigenère square and using the first letter in the plaintext ('W') to select a column. The intersection of that row and column (highlighted below) contains the letter to substitute in the ciphertext for 'W'—in this case, 'T':

	Α	В	C	D	Ε	F	G	Н	I	J	K	L	M	N	0	P	Q	R	S	Т	U	V	W	X	Υ	Z
Α	Α	В	С	D	Ε	F	G	Н	Ι	J	Κ	L	М	N	0	Р	Q	R	S	Т	U	٧	W	Х	Υ	Z
В	В	C	D	Ε	F	G	Н	Ι	J	Κ	L	M	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α
C	C	D	Ε	F	G	Н	Ι	J	Κ	L	M	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В
D	D	Ε	F	G	Н	Ι	J	K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C
Ε	Ε	F	G	Н	Ι	J	K	L	M	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D
F	F	G	Н	Ι	J	K	L	M	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε
G	G	Н	Ι	J	K	L	М	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Е	F
Н	Н	Ι	J	K	L	M	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G
Ι	Ι	J	K	L	M	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н
J	J	K	L	M	N	0	Р	Q	R	S	Т	U	V	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι
K	K	L	M	N	0	Р	Q	R	S	Т	U	V	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J
L	L	M	N	0	Р	Q	R	S	Т	U	V	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K
M	М	N	0	Р	Q	R	S	Т	U	V	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L
N	N	0	Р	Q	R	S	Т	U	V	W	Χ	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M
0	0	Р	Q	R	S	Т	U	V	W	Χ	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	N
P	Р	Q	R	S	Т	U	V	W	Χ	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	N	0
Q	Q	R	S	Т	U	V	W	X	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	N	0	Р
R	R	S	Т	U	V	W	X	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	Ν	0	Р	Q
S	S	Т	U	٧	W	X	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	N	0	Р	Q	R
Т	Т	U	٧	W	X	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	N	0	Р	Q	R	S
U	U	V	W	X	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	N	0	Р	Q	R	S	Т
V	V	W	X	Υ	Z	Α	В	C	D	Е	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U
W	W	Χ	Υ	Z	Α	В	C	D	Е	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	٧
X	Х	Υ	Z	Α	В	C	D	E	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	V	W
Υ	Υ	Ζ	Α	В	C	D	E	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	V	W	X
Z	Z	Α	В	C	D	Е	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	V	W	X	Υ

This process continues for each pair of letters from the secret key and the plaintext:

Plaintext:																			
Repeating key text:	Χ	М	W	U	J	В	٧	Υ	Н	Χ	Z	Χ	M	W	U	J	В	٧	Υ
Ciphertext:	Т	q	h	W	Х	n	z	r	٧	b	m	z	d	u	j	С	j	j	1

Decrypting with the Vigenère Cipher

The decryption process returns the ciphertext to the original plaintext. It's similar to what we described above and requires the same secret key used to encrypt the text. Like the encryption algorithm, the decryption algorithm cycles through the secret key's letters. So, again, we repeat the secret key until the length matches the ciphertext:

Ciphertext:																			
Repeating key text:	Χ	М	W	U	J	В	V	Υ	Н	Χ	Z	Χ	М	W	U	J	В	٧	Υ

We begin the decryption by using the first letter in the repeating key text ('X') to select a row in the Vigenère square. Next, we locate within that row the first letter in the ciphertext ('T'). Finally, we replace the ciphertext letter with the plaintext letter at the top of that column ('W'), as highlighted in the Vigenère square below:

											_		_													
	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	N	0	P	Q	R	S	T	U	٧	W	X	Υ	Z
Α	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ
В	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α
C	C	D	Ε	F	G	Н	Ι	J	Κ	L	M	Ν	0	Р	Q	R	S	Т	U	V	W	Χ	Υ	Ζ	Α	В
D	D	Ε	F	G	Н	Ι	J	Κ	L	M	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C
Ε	Ε	F	G	Н	Ι	J	K	L	Μ	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D
F	F	G	Н	Ι	J	K	L	M	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε
G	G	Н	Ι	J	K	L	М	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F
Н	Н	Ι	J	K	L	M	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G
I	Ι	J	K	L	M	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н
J	J	K	L	М	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι
K	K	L	М	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J
L	L	Μ	Ν	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K
M	M	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L
N	N	0	Р	Q	R	S	Т	U	V	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M
0	0	Р	Q	R	S	Т	U	V	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	Κ	L	M	N
P	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	Κ	L	М	N	0
Q	Q	R	S	Т	U	٧	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	Ν	0	Р
R	R	S	Т	U	V	W	Χ	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	Ν	0	Р	Q
S	S	Т	U	V	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	N	0	Р	Q	R
Т	Т	U	V	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	N	0	Р	Q	R	S
U	U	V	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	Ν	0	Р	Q	R	S	Т
V	V	W	Χ	Υ	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	M	N	0	Р	Q	R	S	Т	U
W	W	Χ	Υ	Ζ	Α	В	C	D	Е	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	V
X	Χ	Υ	Z	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	V	W
Y	Υ	Z	Α	В	C	D	Е	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	٧	W	X
Z	Ζ	Α	В	C	D	Ε	F	G	Н	Ι	J	K	L	М	N	0	Р	Q	R	S	Т	U	V	W	Χ	Υ

This process continues for each pair of letters from the secret key and the ciphertext:

Ciphertext:																			
Repeating key text:	Χ	M	W	U	J	В	٧	Υ	Н	Χ	Z	Χ	M	W	U	J	В	٧	Υ
Plaintext:	W	е	1	С	0	m	е	t	0	е	n	С	r	у	р	t	i	0	n

Implementing the Vigenère Cipher

For this exercise, you should implement your Vigenère cipher code in the file cipher.c. This source-code file should contain the following items:

- Function checkKey receives a secret-key string and returns true if that string consists only of letters. Otherwise, this function returns false, in which case the key cannot be used with the Vigenère cipher algorithm. This function is called by the encrypt and decrypt functions described below.
- Function getSubstitution receives a secret-key character, a character from a plaintext or ciphertext string and a bool indicating whether to encrypt (true) or decrypt (false) the character in the second argument. This function is called by the encrypt and decrypt functions (described below) to perform the Vigenère cipher encryption or decryption algorithm for one character. The function contains the Vigenère square as a 26-by-26 two-dimensional static const char array.
- Function encrypt receives a string containing the plaintext to encrypt, a character array in which to write the encrypted text, and the secret key. The function iterates through the plaintext characters. For each letter, encrypt calls getSubstitution, passing the current secret-key character, the letter to encrypt and true. Function getSubstitution then performs the Vigenère cipher encryption algorithm for that letter and returns its ciphertext equivalent.
- Function decrypt receives a string containing the ciphertext to decrypt, a character array in which to write the resulting plaintext, and the secret key used to create the ciphertext. The function iterates through the ciphertext characters. For each letter, decrypt calls getSubstitution, passing the current secret key character, the letter to decrypt and false. Function getSubstitution then performs the Vigenère cipher decryption algorithm for that letter and returns its plaintext equivalent.

Other Files You Should Create

In addition to cipher.c, you should create the following code files:

- cipher.h should contain the encrypt and decrypt function prototypes.
- **cipher_test.c**, which **#includes "cipher.h"** and uses your encrypt and decrypt functions to encrypt and decrypt text.

cipher_test.c

In your application, perform the following tasks:

1. Prompt for and input a plaintext sentence to encrypt and a secret key consisting only of letters, call encrypt to create the ciphertext, then display it. Use our secret key XMWUJBVYHXZ—this will enable you to decrypt the gibberish at the beginning of this exercise.

- 2. Use your decrypt function and the secret key you entered in *Step 1* to decrypt the ciphertext you just created. Display the resulting plaintext to ensure your decrypt function worked correctly.
- 3. Prompt for and input either the ciphertext section title that precedes this exercise or the exercise ciphertext title. Then, use your **decrypt function** and the **secret-key text** you entered in *Step 1* to **decrypt the ciphertext**.

As always, you should ensure that the character arrays into which you write encrypted or decrypted text are large enough to store the text and its terminating null character.

Once your Vigenère cipher encryption and decryption algorithms work, have some fun sending and receiving encrypted messages with your friends. When you pass your secret key to the person who'll use it to decrypt your ciphertext messages, focus on keeping your key secure.

Compiling Your Code

In Visual C++ and Xcode, simply add all three files to your project, then compile and run the code. For GNU gcc, execute the following command from the folder containing cipher.c, cipher.h and cipher_test.c files:

gcc -std=c18 -Wall cipher.c cipher_test.c -o cipher_test

This will create the command cipher_test, which you can run with ./cipher_test.

Weakness in Secret-Key Cryptography: A Look to Public-Key Cryptography

Secret-key encryption and decryption have a weakness—the ciphertext is only as secure as the secret key. The ciphertext can be decrypted by anyone who discovers or steals the secret key. In the next exercise, we introduce public-key cryptography. This technique performs encryption with a public key known to every sender who may want to send a secret message to a particular receiver. The public key can be used to encrypt messages but not decrypt them. The messages can be decrypted only with a paired private key known only to the receiver, so it's much more secure than the secret key in secret-key cryptography. In the next case study exercise, you'll explore public-key cryptography.

A Note about Cryptography and Computing Power

Ideally, Ciphertext should be impossible to "break"—that is, it should not be possible to determine the plaintext from the ciphertext *without* the decryption key. For various reasons, that goal is impractical. So, designers of cryptography schemes settle for making them extraordinarily difficult to break. One problem with today's increasingly powerful computers is that they're making it possible to break most encryption schemes in use over the last few decades.

Cryptography is at the root of cryptocurrencies such as Bitcoin. The phenomenally powerful computers that quantum computing will make possible are putting

^{9. &}quot;Cryptocurrency." Accessed December 25, 2020. https://www.investopedia.com/terms/c/cryptocurrency.asp.

cryptography schemes and cryptocurrencies at risk. 10,11 The cryptocurrency community is working on these challenges. 12,13,14

8.42 (Vigenère Cipher Modification—Supporting All ASCII Characters) Your Vigenère cipher implementation from Exercise 8.41 encrypts and decrypts only the letters A–Z. All other characters simply pass through as is. Modify your implementation to support the complete ASCII character set shown in Appendix B.

SEC Secure C Programming Case Study: Public-Key Cryptography

8.43 (RSA^{15,16,17} Public-Key Cryptography) In the last case study, you began learning about secret-key cryptography. The sender's plaintext is encrypted with a secret key to form ciphertext. The receiver uses the same secret key to decode the ciphertext, forming the original plaintext—this is called symmetric encryption. A problem with secret-key cryptography is that the security of the ciphertext is only as good as the security of the secret key, and several copies of that key are "floating around." In an attempt to correct this problem, public-key cryptography was proposed by Diffie–Hellman.¹⁸

In this case-study exercise, we walk step-by-step through the RSA Public-Key Cryptography algorithm. In particular, we focus on how to generate:

- the public key that any sender can use to encrypt plaintext into ciphertext for a particular receiver, and
- the private key that only the particular receiver can use to decrypt the ciphertext.

RSA is based on sophisticated mathematics, but the steps you need to perform to generate the public and private keys, encrypt messages with the public key and decrypt

^{10. &}quot;The Impact of Quantum Computing on Present Cryptography." Accessed December 25, 2020. https://arxiv.org/pdf/1804.00200.pdf.

^{11. &}quot;Quantum Computing and its Impact on Cryptography." Accessed December 25, 2020. https://www.cryptomathic.com/news-events/blog/quantum-computing-and-its-impacton-cryptography.

^{12. &}quot;How Should Crypto Prepare for Google's 'Quantum Supremacy'?" Accessed December 25, 2020. https://www.coindesk.com/how-should-crypto-prepare-for-googles-quantumsupremacy.

^{13. &}quot;Here's Why Quantum Computing Will Not Break Cryptocurrencies." Accessed December 25, https://www.forbes.com/sites/rogerhuang/2020/12/21/heres-why-quantumcomputing-will-not-break-cryptocurrencies/.

^{14. &}quot;How the Crypto World Is Preparing for Quantum Computing, Explained." Accessed December 25, 2020. https://cointelegraph.com/explained/how-the-crypto-world-is-preparing-for-quantum-computing-explained.

^{15. &}quot;RSA (cryptosystem)." Accessed January 6, 2021. https://en.wikipedia.org/wiki/ RSA_(cryptosystem).

^{16. &}quot;RSA Algorithm." Accessed January 6, 2021. https://simple.wikipedia.org/wiki/ RSA_algorithm.

^{17. &}quot;PKCS #1: RSA Cryptography Specifications Version 2.2." Accessed January 8, 2021. https:// tools.ietf.org/html/rfc8017.

^{18. &}quot;New Directions in Cryptography." Accessed January 8, 2021. https://ee.stanford.edu/ ~hellman/publications/24.pdf.

messages with the private key are straightforward, as we'll show momentarily. Industrial-quality RSA works with enormous prime numbers consisting of hundreds of digits. To keep our explanations simple and to enable you to quickly build a smallscale working version of RSA, we're going to use only small prime numbers in our explanations. Such small-prime-number RSA versions are not very secure, but they'll help you understand how RSA works.

Public-Key Cryptography

Whitfield Diffie and Martin Hellman, in their paper "New Directions in Cryptography,"19 introduced public-key cryptography to address the weakness of secret-key cryptography—which is the vulnerability of the secret key having to be known by both the sender and the receiver. They came up with the idea but not an implementation of the scheme.

RSA Public-Key Cryptography

Rivest, Shamir and Adelman were the first to publish a working implementation of public-key cryptography. The scheme, called RSA²⁰, bears the initials of their last names. RSA is one of the most widely implemented public-key cryptography schemes in the world.²¹ Because RSA can be slow,²² many organizations prefer to stick to RERF faster private-key encryption, using RSA to securely send the secret key.

Historical Notes

Clifford Cocks in the U.K. created a workable public-key scheme several years before the RSA paper was published, 23 but his work was classified, so it was not revealed until about 20 years after RSA appeared.

The company RSA Security held a patent on the RSA algorithm. In 2000, that patent was coming due for renewal—instead of renewing, they placed the algorithm into the public domain.²⁴

^{19. &}quot;New Directions in Cryptography." Accessed January 8, 2021. https://ee.stanford.edu/ ~hellman/publications/24.pdf.

^{20.} R. Rivest; A. Shamir; L. Adleman (February 1978). "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" (PDF). Communications of the ACM. 21 (2): 120-126. https://people.csail.mit.edu/rivest/Rsapaper.pdf.

^{21. &}quot;RSA algorithm (Rivest-Shamir-Adleman)." Accessed January 8, 2021. https://searchsecurity.techtarget.com/definition/RSA.

^{22. &}quot;RSA (cryptosystem)." Accessed January 6, 2021. https://en.wikipedia.org/wiki/ RSA_(cryptosystem).

^{23. &}quot;Clifford Cocks." Accessed January 8, 2021. https://en.wikipedia.org/wiki/Clifford_Cocks.

^{24. &}quot;RSA Security Releases RSA Encryption Algorithm into Public Domain." Accessed January 8, 2021. https://web.archive.org/web/20071120112201/http://www.rsa.com/press_release.aspx?id=261.

RSA Algorithm Steps

Steps 1–5 below use small integer values to explain how the RSA algorithm generates a public-key/private-key pair. Then, Step 6 uses the public key to encrypt plaintext into ciphertext, and Step 7 uses the private key to decrypt the ciphertext back to the original plaintext. The steps we show are based on the original RSA paper²⁵ and the RSA Algorithm Wikipedia page.²⁶

RSA Algorithm Step I—Choose Two Prime Numbers

Choose two different prime numbers p and q. For this case study, we'll use small prime numbers—p=13 and q=17. This will keep the calculations manageable in our discussions and on your computer using C with its limited-range, built-in integer data types. In commercial-grade RSA cryptography systems, these prime numbers typically are hundreds of digits each and chosen at random. For a sense of how large the integers in RSA can be, visit the RSA Numbers webpage

https://en.wikipedia.org/wiki/RSA_numbers

which shows various integers from 100 to 617 digits in length. The C integer data types int, long int and long long int cannot hold integers this large, so special processing is required to accommodate such large numbers.

RSA Algorithm Step 2—Calculate the Modulus (n), Which Is Part of Both the Public and Private Keys

Calculate the **modulus n**, which is simply the product of p and q:

$$n = p * q$$

Based on p = 13 and q = 17, n is 221. As you'll see, n is part of both the public and private keys. The p and q values are kept private.

RSA Algorithm Step 3—Calculate the Totient Function

Calculate $\Phi(n)$ —pronounced "phi of n"—which is Euler's totient function.²⁷ This is calculated simply as:

$$\Phi(n) = (p - 1) * (q - 1)$$

Given p = 13 and q = 17, $\Phi(n)$ is

$$\Phi(n) = 12 * 16 = 192$$

This number is used in the calculations that determine the encryption exponent (e) and decryption exponent (d), which will help us encrypt plaintext and decrypt ciphertext, respectively, as you'll see below.

^{25.} R. Rivest; A. Shamir; L. Adleman (February 1978). "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" (PDF). Communications of the ACM. 21 (2): 120–126. https://people.csail.mit.edu/rivest/Rsapaper.pdf.

^{26. &}quot;RSA Algorithm." Accessed January 6, 2021. https://simple.wikipedia.org/wiki/RSA_algorithm.

^{27. &}quot;Euler's totient function." Accessed January 7, 2021. https://en.wikipedia.org/wiki/Euler%27s_totient_function.

RSA Algorithm Step 4—Select the Public-Key Exponent (e) for Encryption Calculations

Next, we **choose an exponent**, **e**, **for encryption**, which is subject to the following rules:

- $1 < e < \Phi(n)$
- e must be coprime with $\Phi(n)$.

Two integers are coprime if they have no common factors other than 1.

In our example, the integers that satisfy the first rule for $\Phi(n)$ = 192 are the values 2–191. The prime factorization of 192 is

```
192 = 2 * 2 * 2 * 2 * 2 * 2 * 3
```

The value for e must be coprime with $\Phi(n)$, so we must eliminate from consideration for e any prime factors and all their multiples. Thus, the value 2 and all the other even integers from 2–190 are eliminated, as are the value 3 and all its multiples. This leaves the following odd values as possible values for e:

```
5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 53 55 59 61 65 67 71 73 77 79 83 85 89 91 95 97 101 103 107 109 113 115 119 121 125 127 131 133 137 139 143 145 149 151 155 157 161 163 167 169 173 175 179 181 185 187 191
```

Any of these values can be used as the public encryption key's exponent (e). For our continuing discussion we'll choose 37, so our public key is (37, 221).

RSA Algorithm Step 5—Select the Private-Key Exponent (d) for Encryption Calculations

The final step is to determine the private key's exponent, d, for decryption. We must choose a value for d such that

```
(d * e) \mod \Phi(n) = 1
```

In our example, the first value of d for which this is true is 109. We can check whether the preceding calculation produces 1 by plugging in the values of d, e and $\Phi(n)$:

```
(109 * 37) mod 192
```

The value of 109 * 37 is 4033. If you multiply 192 by 21, the result is 4032, leaving a remainder of 1. So, 109 is a valid value for d. There are many potential values of d—each is 109 plus a multiple of the totient (192). For instance, 301 (109 + 1 * 192):

```
(301 * 37) mod 192
```

301 * 37 is 11137, which has the remainder 1 when divided by 192—192 * 58 is 11136, leaving a remainder of 1. So values for d such as the following will work:

```
109 301 493 685 877 ...
```

We chose 109, so our private key is (109, 221).

Encrypting a Message with RSA

Once you have the public key, it's easy to encrypt a message using RSA. Given a plaintext integer message (M) to encrypt into ciphertext (C) and a public key consist-

ing of two positive integers e (for encrypt) and n—commonly represented as (e, n)—a message sender can encrypt M with the calculation:

$$C = M^e \mod n$$

The value of M must be in the range 0 <= M < n. Otherwise, you must break the message into values within that range and encrypt each separately.

Let's encrypt the M value 122 using our public key (37, 221):

$$C = 122^{37} \mod 221$$

The value 122³⁷ is an enormous number, but you can perform this calculation using the Wolfram Alpha website at

```
https://www.wolframalpha.com/input/
```

Enter the calculation as follows (the ^ represents exponentiation in Wolfram Alpha):

You'll see that the result is 5, which is our ciphertext.

Decrypting a Message with RSA

It's also easy to decrypt a message if you have the private key. Given a ciphertext integer message (C) to decrypt into the original plaintext message (M) and a private key consisting of two positive integers d and n—commonly represented as (d, n)—a message receiver can decrypt C with the following calculation:

$$M = C^d \mod n$$

Let's decrypt the C value 5 using our public key (109, 221):

$$C = 5^{109} \mod 221$$

Once again, the value 5¹⁰⁹ is an enormous number, but you can perform this calculation using Wolfram Alpha by entering the calculation as follows:

```
5^109 mod 221
```

You'll see that the result is 122, which is our plaintext.

Note that n is part of *both* the public key and the private key. You'll also see that the exponent d's value is based on the exponent e and the modulus value n.

Encrypting and Decrypting Strings

Suppose you wish to use RSA to encrypt a plaintext message, such as

```
Damn the torpedoes, full speed ahead!28
```

As you know, the RSA algorithm encrypts *only* integer messages in the range 0 <= M < n. To encrypt the preceding message, you must map the characters to integer values.

One way to convert characters to integers is to use each character's numeric value in the underlying character set. For this exercise, assume ASCII characters, which have integer values in the range 0–127 (see Appendix B). Provided that a character's

^{28.} David Glasgow Farragut—an American Civil War Union officer and the first full admiral in the U.S. Navy. Accessed January 8, 2021. https://en.wikipedia.org/wiki/David_Farragut.

integer value is less than n, you can encrypt that value as shown previously. You can store each resulting ciphertext integer in an integer array. If you try to display those ciphertext integers as characters, you may see some strange symbols. For instance, the ciphertext integers may represent special characters, such as newlines or tabs, or may be outside the ASCII range. When you decrypt the ciphertext, you can take each resulting integer, cast it to a char, then place it into a char array that will represent the deciphered plaintext. Be sure to null-terminate your string before displaying it.

Programming the RSA Algorithm

Now, implement the RSA algorithm in C. Enable the user to encrypt and decrypt a simple integer, then encrypt and decrypt a line of text. Your program should produce an output dialog similar to the following:

```
Enter a prime number for p: 13
Enter a prime number for q: 17
n is 221
totient is 192
Candidates for e: 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 53 55 59 61 65
67 71 73 77 79 83 85 89 91 95 97 101 103 107 109 113 115 119 121 125 127 131 133
137 139 143 145 149 151 155 157 161 163 167 169 173 175 179 181 185 187 191
Select a value for e from the preceding candidates: 37
Candidate for d: 109
Select a value for d--either the d candidate above
or d plus a multiple of the totient: 109
Enter a non-negative integer less than n to encrypt: 122
The ciphertext is: 5
The decrypted plaintext is: 122
Enter a sentence to encrypt:
Damn the torpedoes, full speed ahead!
The ciphertext is:
DG'
;X}eW;es9 fh s}eeW GueGW!
The decrypted plaintext is:
Damn the torpedoes, full speed ahead!
```

As you implement the RSA algorithm, keep the following hints in mind:

Modular exponentiation: Raising a plaintext message to a large exponent (e.g., 122³⁷) results in enormous values that C's limited-range, built-in integer types cannot represent. As you know, RSA encryption and decryption calculations perform both exponentiation and modulus operations. These can be combined using modular exponentiation to keep the RSA encryption and decryption

calculations within manageable ranges. Define a function named modular Pow that performs modular exponentiation. For the modular exponentiation algorithm, see the psuedocode at

```
https://en.wikipedia.org/wiki/Modular_exponentiation#Memory-efficient_method
```

- Calculating the greatest common divisor: The candidate values for e (*Step 4*) must be coprime with the totient—again, their only common factor is 1. To determine if two numbers are coprime, you'll need a function gcd that calculates the greatest common divisor of two integers. Your program should display all possible candidate values for e. You were asked to write a gcd function in Exercise 5.29.
- Checking for prime numbers—The RSA algorithm requires two prime numbers, p and q. You should define a function isPrime that determines if an integer is indeed a prime number—use it to confirm that the p and q values the user entered are prime. Exercise 6.30 asked you to implement the Sieve of Eratosthenes to find prime values.

Your program also should define the following functions:

- A function to encrypt a plaintext message M using the public key (e, n):
 int encrypt(int M, int e, int n);
- A function to decrypt a ciphertext message C using the private key (d, n): int decrypt(int C, int d, int n);
- A function to encrypt a string by calling the encrypt function for each character of the string and placing the results into an integer array:

```
void encryptString(
    char* plaintext, int ciphertext[], int e, int n);
```

• A function to decrypt ciphertext from an integer array by calling the decrypt function for each integer and placing the results into a char array. The size parameter represents the number of characters that were encrypted. Be sure to terminate the string in decryptedPlaintext with a null character ('\0'):

```
void decryptString(int ciphertext[],
   char decryptedPlaintext[], size_t size, int d, int n);
```

References

For a nice video explanation of the RSA algorithm, see the following two-part video presentation:

- The RSA Encryption Algorithm (1 of 2: Computing an Example):²⁹ https://www.youtube.com/watch?v=4zahvcJ9glg
- The RSA Encryption Algorithm (2 of 2: Generating the Keys):³⁰

^{29.} Woo, Eddie (misterwootube). "The RSA Encryption Algorithm (1 of 2: Computing an Example)," November 4, 2014. https://www.youtube.com/watch?v=4zahvcJ9glg.

https://www.youtube.com/watch?v=o0cTVTpUsPQ

8.44 (An Improvement to the RSA Algorithm) In 1998, an improvement was made to the RSA algorithm replacing $\Phi(n)$ with $\lambda(n)$ (pronounced "lambda of n"):^{31,32}

$$\lambda(n) = 1cm((p - 1), (q - 1))$$

where 1cm represents the least common multiple.³³ We used $\Phi(n)$ in the previous RSA exercise with p = 13 and q = 17. The corresponding new $\lambda(n)$ calculation would be

$$\lambda(n) = 1cm(12, 16)$$

where the least common multiple of 12 and 16 is 48, as you can see in the lists of multiples below:

```
12 24 36 48 ...
16 32 48 60 ...
```

Make a copy of your code solution for the previous exercise and replace each use of $\Phi(n)$ with $\lambda(n)$, then test your updated code with the same prime-number values for p and q. When you encrypt the plaintext using the $\lambda(n)$ approach, your ciphertext will likely be different, but the decrypted plaintext should be the same.

- **8.45** (Stress Testing Your RSA Algorithm's Limits) Try your program with gradually increasing values for p and q. How large do they get before the program no longer works? Also, test your program with increasingly larger candidates for e and d.
- **8.46** (Enhancing Your RSA Code) Modify your RSA program as follows:
 - a) Your program displayed all the possible candidates for the encryption exponent e. Modify your program to show the first five potential values for the decryption exponent d (i.e., the first value of d plus 1 * totient, the first value of d plus 2 * totient, etc.). Follow your list of possibilities with an ellipsis (...).
 - b) As your prime numbers p and q get larger, you'll eventually surpass the int type's maximum value limit, which you can find in <1imits.h>. Modify your code to do all RSA integer calculations using type long long int. Note that even that type will be inadequate for holding the enormous integers you'd use in industrial quality RSA. It would require special programming to use such larger integer values. Remember to change any printf and scanf statements' %d conversion specifiers to %11d.

^{30.} Woo, Eddie (misterwootube). "The RSA Encryption Algorithm (2 of 2: Generating the Keys)," November 4, 2014. https://www.youtube.com/watch?v=o0cTVTpUsPQ.

^{31. &}quot;RSA Algorithm." Accessed January 7, 2021. https://simple.wikipedia.org/wiki/RSA_algorithm.

^{32. &}quot;PKCS #1: RSA Cryptography Specifications, Version 2.0." Accessed January 7, 2021. https://tools.ietf.org/html/rfc2437.

^{33. &}quot;Least common multiple." Accessed January 7, 2021. https://en.wikipedia.org/wiki/Least_common_multiple.

- **8.47** (Challenge Project: The RSA Problem³⁴) In this exercise, you'll research attacks that have been perpetrated on industrial-strength RSA implementations. You'll then try your own hand at cracking RSA ciphertext created by the small-scale RSA implementation you built in Exercise 8.43. Again, such small-scale implementations are not secure.
 - a) Research the kinds of attacks that have been perpetrated against industrialstrength RSA systems. Note which kinds have succeeded and which have failed.
 - b) RSA's strength comes from the enormous prime numbers p and q (each typically hundreds of digits) used to calculate the far more enormous value of n (which is p * q) and the computational expense of factoring n to find p and q. The "RSA Problem" is the task of decrypting ciphertext given only the public key (e, n). This requires you to find n's prime factors p and q from which you would then derive d and decrypt the ciphertext.

Assume you have a public key (e, n) and ciphertext that was encrypted using that key with your small-scale RSA implementation, but you do not know the private key required to decrypt the ciphertext. Use brute-force computing techniques to find n's prime factors p and q. Then, do the calculations necessary to recover d and decrypt the message.

^{34. &}quot;RSA Problem." Accessed January 8, 2021. https://en.wikipedia.org/wiki/RSA_problem.

Formatted Input/Output



9

Objectives

In this chapter, you'll:

- Use input and output streams.
- Use print formatting capabilities.
- Use input formatting capabilities.
- Print integers, floating-point numbers, strings and characters.
- Print with field widths and precisions.
- Use formatting flags in the printf format control string.
- Output literals and escape sequences.
- Read formatted input using scanf.

- 9.1 Introduction
- 9.2 Streams
- **9.3** Formatting Output with **printf**
- 9.4 Printing Integers
- **9.5** Printing Floating-Point Numbers
 - 9.5.1 Conversion Specifiers e, E and f
 - 9.5.2 Conversion Specifiers g and G
 - 9.5.3 Demonstrating Floating-Point Conversion Specifiers
- **9.6** Printing Strings and Characters
- 9.7 Other Conversion Specifiers
- **9.8** Printing with Field Widths and Precision
 - 9.8.1 Field Widths for Integers
 - 9.8.2 Precisions for Integers, Floating-Point Numbers and Strings
 - 9.8.3 Combining Field Widths and Precisions

9.9 printf Format Flags

- 9.9.1 Right- and Left-Alignment
- 9.9.2 Printing Positive and Negative Numbers with and without the + Flag
- 9.9.3 Using the Space Flag
- 9.9.4 Using the # Flag
- 9.9.5 Using the O Flag
- **9.10** Printing Literals and Escape Sequences
- **9.11** Formatted Input with **scanf**
 - 9.11.1 scanf Syntax
 - 9.11.2 **scanf** Conversion Specifiers
 - 9.11.3 Reading Integers
 - 9.11.4 Reading Floating-Point Numbers
 - 9.11.5 Reading Characters and Strings
 - 9.11.6 Using Scan Sets
 - 9.11.7 Using Field Widths
 - 9.11.8 Skipping Characters in an Input Stream
- **9.12** Secure C Programming

Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises

9.1 Introduction

Presenting results is an important part of the solution to any problem. This chapter discusses in-depth **printf** and **scanf** formatting features, which output data to the standard output stream and input data from the standard input stream. Include the header **<stdio.h>** in programs that call these functions. Chapter 11 discusses several additional functions included in the standard input/output (**<stdio.h>**) library.

9.2 Streams

Input and output are performed with sequences of bytes called streams:

- In input operations, the bytes flow into main memory from a device, such as a keyboard, a solid-state drive, a network connection, and so on.
- In output operations, bytes flow from main memory to a device, such as a computer's screen, a printer, a solid-state drive, a network connection, and so on.

When program execution begins, the program has access to three streams:

- the standard input stream, which is connected to the keyboard,
- the standard output stream, which is connected to the screen, and
- the standard error stream, which also is connected to the screen.

Operating systems allow these streams to be redirected to other devices. Chapter 11 discusses stream-processing in detail.

Self Check

- I (Fill-In) You can _____ the standard streams to other devices. Answer: redirect.
- **2** (*Multiple Choice*) Which of the following statements is *false*?
 - a) Input and output are performed with arrays, which are sequences of bytes.
 - b) In input operations, the bytes flow from a device to main memory.
 - c) In output operations, bytes flow from main memory to a device.
- d) When execution begins, the standard streams are connected to the program. **Answer:** a) is *false*. Actually, input and output are performed with streams, which are sequences of bytes.

9.3 Formatting Output with printf

Throughout the book, you've seen various printf output formatting features. Every printf call contains a format control string that describes the output format. The format control string consists of conversion specifiers, flags, field widths, precisions and literal characters. Together with the percent sign (%), these form conversion specifications. Function printf can perform the following formatting capabilities:

- 1. Rounding floating-point values to an indicated number of decimal places.
- 2. Aligning columns of numbers at their decimal points.
- **3.** right-aligning and left-aligning outputs.
- 4. Inserting literal characters at precise locations in a line of output.
- 5. Representing floating-point numbers in exponential format.
- **6.** Representing unsigned integers in octal and hexadecimal format. Online Appendix E discusses octal and hexadecimal values.
- 7. Displaying data with fixed-size field widths and precisions.

The printf function has the form

printf(format-control-string, other-arguments);

The *format-control-string* describes the output format, and the optional *other-arguments* correspond to the *format-control-string*'s conversion specifications. Every conversion specification begins with a percent sign (%) and ends with a conversion specifier. There can be many conversion specifications in one format control string.

✓ Self

Self Check

- (*Fill-In*) Every printf call contains a _____ that describes the output format. Answer: format control string.
- **2** (*Multiple Choice*) Which of the following is a formatting capability function printf can perform?
 - a) Rounding floating-point values to an indicated number of decimal places, and aligning a column of numbers at their decimal points.

- b) Representing floating-point numbers in exponential format. Representing unsigned integers in octal and hexadecimal format.
- c) Displaying all types of data with fixed-size field widths and precisions.
- d) All of the above are printf formatting capabilities.

Answer: d.

9.4 Printing Integers

An integer is a whole number, such as 776, 0 or -52. Integer values are displayed in one of several formats described by the following integer conversion specifiers.

Conversion specifier	Description
d	Display as a signed decimal integer.
i	Display as a signed decimal integer.
0	Display as an unsigned octal integer.
u	Display as an unsigned decimal integer.
x or X	Display as an unsigned hexadecimal integer. X uses the digits 0-9 and the uppercase letters A-F, and x uses the digits 0-9 and the lowercase letters a-f.
h, 1 or 11 (letter "ell")	These length modifiers are placed before any integer conversion specifier to indicate that the value to display is a short, long or long long integer.

Figure 9.1 prints an integer using each integer conversion specifier. Note that plus signs do not display by default, but we'll show later how to force them to display. Lines 10–11 use the hd and 1d conversion specifiers to display short and 1ong integer values. The L suffix on the literal 200000000L indicates that its type is 1ong—C treats whole-number literals as int. Printing a negative value with a conversion specifier that expects an unsigned value is a logic error. When line 14 displays -455 with %u, the result is the unsigned value 4294966841. A small negative value displays as a large positive integer due to the value's "sign bit" in the underlying binary representation. See online Appendix E for a discussion of the binary number system and the sign bit.

```
// fig09_01.c
// Using the integer conversion specifiers
#include <stdio.h>

int main(void) {
   printf("%d\n", 455);
   printf("%i\n", 455); // i same as d in printf
   printf("%d\n", +455); // plus sign does not print
   printf("%d\n", -455); // minus sign prints
```

Fig. 9.1 Using the integer conversion specifiers. (Part 1 of 2.)

```
\label{eq:printf} $$ printf("%hd\n", 32000); // print as type short $$ printf("%ld\n", 2000000000L); // print as type long $$ printf("%o\n", 455); // octal $$ $$ $$ $$
10
11
12
          printf("%u\n", 455);
printf("%u\n", -455);
13
14
           printf("%x\n", 455); // hexadecimal with lowercase letters
15
          printf("%X\n", 455); // hexadecimal with uppercase letters
16
17 }
455
455
455
-455
32000
2000000000
707
455
4294966841
1c7
1C7
```

Fig. 9.1 Using the integer conversion specifiers. (Part 2 of 2.)

Self Check

- **I** (*Multiple Choice*) Which integer conversion specifier is described by "Display as an unsigned decimal integer"?
 - a) ud.
 - b) ui.
 - c) u.
 - d) None of the above.

Answer: c.

2 (What Does This Code Do?) Show precisely what the following code prints:

```
printf("%d\n", 235);
printf("%i\n", 235);
printf("%d\n", +235);
printf("%d\n", -235);
```

Answer:

235

235

235

-235

9.5 Printing Floating-Point Numbers

Floating-point values contain a decimal point, as in 33.5, 0.0 or -657.983. Floating-point values are displayed using the conversion specifiers summarized below.

Conversion specifier	Description
e or E	Display a floating-point value in exponential notation.
f or F	Display floating-point values in fixed-point notation.
g or G	Display a floating-point value in either the fixed-point form f or the exponential form e (or E), based on the value's magnitude.
L	Place this length modifier before any floating-point conversion specifier to indicate that a long double floating-point value should be displayed.

Exponential Notation

The conversion specifiers e and E display floating-point values in exponential notation—the computer equivalent of scientific notation used in mathematics. For example, the value 150.4582 is represented in scientific notation as

 1.504582×10^{2}

and in exponential notation as

1.504582E+02

In this notation, the E stands for "exponent" and indicates that 1.504582 is multiplied by 10 raised to the second power (E+02).

9.5.1 Conversion Specifiers e, E and f

Values displayed with the conversion specifiers e, E and f show six digits of precision to the decimal point's right by default (e.g., 1.045927). You can specify other precisions explicitly. Conversion specifier f always prints at least one digit to the left of the decimal point, so fractional values will be preceded by "0.". Conversion specifiers e and E precede the exponent with lowercase e or uppercase E. Each prints exactly one digit to the decimal point's left.

9.5.2 Conversion Specifiers g and G

Conversion specifier g (or G) prints in either e (E) or f format with no trailing zeros, so 1.234000 displays as 1.234. The conversion specifier g uses the e (E) format if, after conversion to exponential notation, the value's exponent is less than -4, or the exponent is greater than or equal to the specified precision. Otherwise, g uses the conversion specifier f to print the value. The default precision is six significant digits for g and G—a maximum of six digits will display.

At least one decimal digit is required for the decimal point to be output. For example, the values 0.0000875, 8750000.0, 8.75 and 87.50 are printed as 8.75e-05, 8.75e+06, 8.75 and 87.5 with the conversion specifier g. The value 0.0000875 uses e notation because, when it's converted to exponential notation, its exponent (-5) is less than -4. The value 8750000.0 uses e notation because its exponent (6) is equal to the default precision.

Precision

For conversion specifiers g and G, the precision indicates the maximum number of significant digits to display, including the digit to the left of the decimal point. So, the value 1234567.0 displays as 1.23457e+06, using conversion specification %g. Remember that all floating-point conversion specifiers have a default precision of 6. There are six significant digits in the result—1 to the left of the decimal point and 23457 to the right. For exponential notation, g and G precede the exponent with a lowercase e or uppercase E. When displaying data, make it clear to users whether the data may be imprecise due to formatting, such as rounding errors from specifying precisions.

9.5.3 Demonstrating Floating-Point Conversion Specifiers

Figure 9.2 demonstrates each of the floating-point conversion specifiers. The **%E**, **%e** and **%g** conversion specifications perform rounding, but **%f** does not.

```
// fig09_02.c
   // Using the floating-point conversion specifiers
2
3
   #include <stdio.h>
5
   int main(void) {
6
       printf("%e\n", 1234567.89);
7
       printf("%e\n", +1234567.89); // plus does not print
       printf("%e\n", -1234567.89); // minus prints
printf("%E\n", 1234567.89);
8
9
       printf("%f\n", 1234567.89); // six digits to right of decimal point
10
       printf("%g\n", 1234567.89); // prints with lowercase e
11
12
       printf("%G\n", 1234567.89); // prints with uppercase E
13
    }
1.234568e+06
1.234568e+06
-1.234568e+06
1.234568E+06
1234567.890000
1.23457e+06
1.23457E+06
```

Fig. 9.2 Using the floating-point conversion specifiers.

Self Check

- (*Fill-In*) The conversion specifiers e and E display floating-point values in exponential notation—the computer equivalent of _____ used in mathematics. Answer: scientific notation.
- (Multiple Choice) Which statement about conversion specifiers e, E and f is false?a) Values displayed with the conversion specifiers e, E and f show six digits of

precision to the decimal point's right by default.

- b) Conversion specifier f always prints exactly one digit to the left of the decimal point.
- c) Conversion specifiers e and E print lowercase e and uppercase E, respectively, preceding the exponent, and exactly one digit to the left of the decimal point.
- d) All of the above statements are true.

Answer: b) is *false*. Actually, conversion specifier f prints at least one digit to the left of the decimal point.

9.6 Printing Strings and Characters

The c and s conversion specifiers are used to print individual characters and strings, respectively. Conversion specifier c requires a char argument. Conversion specifier s requires a pointer to char as an argument. Conversion specifier s prints characters until a terminating null ('\0') character is encountered. If the string does not have a null terminator, the result is undefined—printf will either continue printing until it encounters a zero byte or the program will terminate prematurely (i.e., "crash") and ERR \(\subseteq \) indicate a "segmentation fault" or "access violation" error. The program in Fig. 9.3 displays characters and strings with conversion specifiers c and s.

```
I // fig09_03.c
2 // Using the character and string conversion specifiers
3 #include <stdio.h>
4
5 int main(void) {
       char character = 'A'; // initialize char
6
       printf("%c\n", character);
7
8
       printf("%s\n", "This is a string");
9
10
       char string[] = "This is a string"; // initialize char array
11
       printf("%s\n", string);
12
13
       const char *stringPtr = "This is also a string"; // char pointer
14
       printf("%s\n", stringPtr);
15
   }
16
This is a string
This is a string
This is also a string
```

Fig. 9.3 Using the character and string conversion specifiers.

Errors in Format Control Strings

Most compilers do not catch format-control-string errors. You'll typically become aware of such errors when a program fails or produces incorrect results at runtime.



• Using %c to print a string is a logic error—%c expects a char argument. A string is a pointer to char (i.e., a char *).

Using %s to print a char argument usually causes a fatal execution-time logic 💢 ERR error called an access violation. The conversion specification %s expects an argument of type pointer to char, so it treats the char's numeric value as a pointer. Such small numeric values often represent memory addresses that are restricted by the operating system.

Self Check

(Fill-In) Conversion specifier s causes characters to be printed until a ______ is encountered.

Answer: terminating null ('\0') character.

2 (*True*/*False*) Compilers catch errors in the format-control string, so you will not experience incorrect results at runtime.

Answer: False. Actually, most compilers do not catch errors in the format-control string. You typically will not become aware of such errors until a program fails or produces incorrect results at runtime.

9.7 Other Conversion Specifiers

Consider the p and % conversion specifiers:

- p—Displays a pointer value in an implementation-defined manner.
- —Displays the percent character.

Figure 9.4's p prints ptr's value and x's address in an implementation-defined manner, typically using hexadecimal notation. Variables ptr and x have identical values because line 7 assigns x's address to ptr. The addresses displayed on your system will vary. The last printf statement uses **%** to display the **%** character—**%** is required because printf normally treats % as the beginning of a conversion specification. Try- (X) ERR ing to display a literal percent character using % rather than %% in the format control string is an error. When % appears in a format control string, it must be followed by a conversion specifier.



```
// fig09_04.c
   // Using the p and % conversion specifiers
3
  #include <stdio.h>
5
  int main(void) {
6
     int x = 12345;
7
     int *ptr = &x;
8
9
     printf("The value of ptr is %p\n", ptr);
10
     printf("The address of x is p\n\n", &x);
11
12
     13
   }
```

Using the p and % conversion specifiers. (Part 1 of 2.)

```
The value of ptr is 0x7ffff6eb911c
The address of x is 0x7ffff6eb911c
Printing a % in a format control string
```

Fig. 9.4 Using the p and % conversion specifiers. (Part 2 of 2.)



Self Check

- (Fill-In) A printf statement uses ______ to print the % character. Answer: %%.
- 2 (True/False) The conversion specifier p displays an address in decimal notation. **Answer:** False. Actually, the conversion specifier p displays an address in an implementation-defined manner—typically, using hexadecimal notation.

9.8 Printing with Field Widths and Precision

The exact size of a field in which data is printed is specified by a field width. If the field width is larger than the data being printed, the data will normally be rightaligned within that field. An integer representing the field width is inserted between the percent sign (%) and the conversion specifier (e.g., %4d).

9.8.1 Field Widths for Integers

Figure 9.5 prints two groups of five numbers each, right-aligning those numbers containing fewer digits than the field width. Values wider than the field still display in full. Note that the minus sign for a negative value uses one character position in the field width. Field widths can be used with all conversion specifiers. Not providing a ERR (X) sufficiently large field width to handle a printed value can offset other data being printed, producing confusing outputs. Know your data!

```
I
   // fig09_05.c
 2 // Right-aligning integers in a field
 3 #include <stdio.h>
 4
 5 int main(void) {
        printf("%4d\n", 1);
printf("%4d\n", 12);
 6
 7
        printf("%4d\n", 123);
 8
        printf("%4d\n", 1234);
 9
10
        printf("%4d\n\n", 12345);
11
        printf("%4d\n", -1);
12
13
        printf("%4d\n", -12);
14
        printf("%4d\n", -123);
        printf("%4d\n", -1234);
printf("%4d\n", -12345);
15
16
17
    }
```

Fig. 9.5 Right-aligning integers in a field. (Part 1 of 2.)

```
1
12
123
1234
12345

-1
-12
-123
-1234
-12345
```

Fig. 9.5 Right-aligning integers in a field. (Part 2 of 2.)

9.8.2 Precisions for Integers, Floating-Point Numbers and Strings

Function printf also enables you to specify the precision with which data is printed. Precision has different meanings for different types:

- When used with integer conversion specifiers, precision indicates the minimum number of digits to be printed. If the printed value contains fewer digits than the specified precision and the precision value has a leading zero or decimal point, zeros are prefixed to the printed value until the total number of digits is equivalent to the precision. If neither a zero nor a decimal point is present in the precision value, spaces are inserted instead. The default precision for integers is 1.
- When used with floating-point conversion specifiers e, E and f, the precision is the number of digits to appear after the decimal point.
- When used with conversion specifiers g and G, the precision is the maximum number of significant digits to be printed.
- When used with conversion specifier s, the precision is the maximum number of characters written from the beginning of the string.

To use precision, place a decimal point (.), followed by an integer representing the precision between the percent sign and the conversion specifier. Figure 9.6 demonstrates the use of precision in format control strings. When you print a floating-point value with a precision smaller than the value's original number of decimal places, it's rounded.

```
// fig09_06.c
// Printing integers, floating-point numbers and strings with precisions
#include <stdio.h>

int main(void) {
   puts("Using precision for integers");
   int i = 873; // initialize int i
   printf("\t%.4d\n\t%.9d\n\n", i, i);
```

Fig. 9.6 Printing integers, floating-point numbers and strings with precisions. (Part 1 of 2.)

```
9
       puts("Using precision for floating-point numbers");
10
       double f = 123.94536; // initialize double f
11
       printf("\t^{\infty}.3f\n\t^{\infty}.3e\n\t^{\infty}.3g\n\n", f, f, f);
12
13
       puts("Using precision for strings");
14
       char s[] = "Happy Birthday"; // initialize char array s
15
       printf("\t^{11s}\n", s);
16
17
    }
Using precision for integers
         0873
         000000873
Using precision for floating-point numbers
        123.945
        1.239e+02
         124
Using precision for strings
         Happy Birth
```

Fig. 9.6 Printing integers, floating-point numbers and strings with precisions. (Part 2 of 2.)

9.8.3 Combining Field Widths and Precisions

The field width and the precision can be combined by placing the field width, followed by a decimal point, followed by a precision between the percent sign and the conversion specifier, as in the statement

```
printf("%9.3f", 123.456789);
```

which displays 123.457 with three digits to the right of the decimal point rightaligned in a nine-digit field.

Specifying Field Widths and Precisions As Arguments

It's possible to specify the field width and the precision using integer expressions in the argument list following the format control string. To use this feature, insert an asterisk (*) in place of the field width or precision (or both). The matching int argument in the argument list is evaluated and used in place of the asterisk. A field width's value may be either positive or negative (which causes the output to be left-aligned in the field, as described in the next section). The statement

```
printf("%*.*f", 7, 2, 98.736);
```

uses 7 for the field width, 2 for the precision and outputs the value 98.74 right-aligned.

Self Check

- [Multiple Choice] Which of the following statements is false?
 - a) The default precision for integers is 1.

- b) When used with floating-point conversion specifiers e, E and f, the precision is the number of digits to appear before the decimal point.
- c) When used with conversion specifiers g and G, the precision is the maximum number of significant digits to be printed.
- d) When used with conversion specifier s, the precision is the maximum number of characters written from the beginning of the string.

Answer: b) is *false*. When used with floating-point conversion specifiers e, E and f, the precision is the number of digits to appear after the decimal point.

2 (What Does This Code Do?) Describe precisely what the following code prints: printf("%9.3f", 123.456789);

Answer: The code right-aligns in a nine-digit field the rounded value 123.457.

3 (What Does This Code Do?) Describe precisely what the following code prints: printf("%*.*f", 7, 2, 98.736);

Answer: The statement uses 7 for the field width, 2 for the precision and outputs the value 98.74 right-aligned in a field of 7.

9.9 printf Format Flags

Function printf also provides *flags* to supplement its output formatting capabilities. The following table summarizes the five flags you can use in format control strings.

Flag	Description
- (minus sign)	Left-align the output within the specified field.
+	Display a plus sign preceding positive values and a minus sign preceding negative values.
space	Print a space before a positive value not printed with the + flag.
#	Prefix 0 to the output value when used with the octal conversion specifier o.
	Prefix 0x or 0X to the output value when used with the hexadecimal conversion specifiers x or X.
	Force a decimal point for a floating-point number printed with e, E, f, g or G that does not contain a fractional part. Normally, the decimal point is printed only if a digit follows it. For g and G specifiers, trailing zeros are not eliminated.
0 (zero)	Pad a field with leading zeros.

9.9.1 Right- and Left-Alignment

Flags in a conversion specification are placed immediately to the right of the % and before the format specifier. Several flags may be combined in one conversion specifier. Figure 9.7 demonstrates right-alignment and left-alignment of a string, an integer, a character and a floating-point number. Lines 6 and 8 output lines of numbers representing the column positions, so you can confirm that the right- and left-alignment worked correctly.

```
// fig09_07.c
   // Right- and left-aligning values
3 #include <stdio.h>
4
5 int main(void) {
      puts("1234567890123456789012345678901234567890");
6
7
      printf("%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23);
      puts("1234567890123456789012345678901234567890");
8
      printf("%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23);
9
10
   }
1234567890123456789012345678901234567890
     hello
                  7
                            a 1.230000
1234567890123456789012345678901234567890
hello 7
```

Fig. 9.7 Right- and left-aligning values.

9.9.2 Printing Positive and Negative Numbers with and without the + Flag

Figure 9.8 prints a positive number and a negative number, each with and without the + flag. The minus sign is displayed in both cases, but the plus sign is displayed only when the + flag is used.

```
1  // fig09_08.c
2  // Printing positive and negative numbers with and without the + flag
3  #include <stdio.h>
4
5  int main(void) {
6    printf("%d\n%d\n", 786, -786);
7    printf("%+d\n%+d\n", 786, -786);
8 }
```

Fig. 9.8 Printing positive and negative numbers with and without the + flag.

9.9.3 Using the Space Flag

Figure 9.9 prefixes a space to the positive number with the space flag. This is useful for aligning positive and negative numbers with the same number of digits. The value -547 is not preceded by a space in the output because of its minus sign.

```
1  // fig09_09.c
2  // Using the space flag
3  // not preceded by + or -
4  #include <stdio.h>
5
6  int main(void) {
7    printf("% d\n% d\n", 547, -547);
8 }
```

Fig. 9.9 | Using the space flag.

9.9.4 Using the # Flag

Figure 9.10 uses the # flag to prefix 0 to the octal value and 0x and 0X to the hexadecimal values. For g, it forces the decimal point to print.

```
I // fig09_10.c
2 // Using the # flag with conversion specifiers
   // o, x, X and any floating-point specifier
  #include <stdio.h>
5
  int main(void) {
6
7
       int c = \frac{1427}{7}; // initialize c
8
       printf("%#o\n", c);
       printf("%#x\n", c);
9
10
       printf("%#X\n", c);
П
12
       double p = 1427.0; // initialize p
13
       printf("\n%g\n", p);
14
       printf("%#g\n", p);
15 }
02623
0x593
0X593
1427
1427.00
```

Fig. 9.10 Using the # flag with conversion specifiers.

9.9.5 Using the 0 Flag

Figure 9.11 combines the + flag and the 0 (zero) flag to print 452 in a nine-space field with a + sign and leading zeros, then prints 452 again using only the 0 flag and a nine-space field.

```
I // fig09_11.c
2 // Using the 0 (zero) flag
3 #include <stdio.h>
4
5 int main(void) {
6    printf("%+09d\n", 452);
7    printf("%09d\n", 452);
8 }
+00000452
000000452
```

Fig. 9.11 Using the 0 (zero) flag.

Self Check

- I (Multiple Choice) Which printf format flag is described by, "display a plus sign preceding positive values and a minus sign preceding negative values"?
 - a) -.
 - b) +.
 - c) 0).
 - d) None of the above.

Answer: b.

2 (What Does This Code Do?) Show precisely what the following code prints:

Answer:

```
123456789012345678901234567890

C18 9 g 6.410000

123456789012345678901234567890

C18 9 g 6.410000
```

3 (What Does This Code Do?) Show precisely what the following code prints:

```
printf("%d\n%d\n", 437, -437);
printf("%+d\n%+d\n", 437, -437);
```

Answer:

437

-437

+437

-437

9.10 Printing Literals and Escape Sequences

As you've seen throughout the book, literal characters included in the format control string are simply output by printf. However, there are several "problem" characters,

such as the *quotation mark* (") that delimits the format control string itself. Various control characters, such as *newline* and *tab*, must be represented by escape sequences. An escape sequence is represented by a backslash (\), followed by a particular escape character. The following table lists the escape sequences and the actions they cause.

Escape sequence	Description
\' (single quote)	Output the single quote (') character.
\" (double quote)	Output the double quote (") character.
\? (question mark)	Output the question mark (?) character.
\\ (backslash)	Output the backslash (\) character.
\a (alert or bell)	Cause an audible (bell) or visual alert (typically, flashing the window in which the program is running).
\b (backspace)	Move the cursor back one position on the current line.
\f (new page or form feed)	Move the cursor to the next logical page's start.
\n (newline)	Move the cursor to the beginning of the <i>next</i> line.
\r (carriage return)	Move the cursor to the beginning of the <i>current</i> line.
\t (horizontal tab)	Move the cursor to the next horizontal tab position.
\v (vertical tab)	Move the cursor to the next vertical tab position.

Self Check

- [(Multiple Choice) Which of the following statements a), b) or c) is false?
 - a) Literal characters included in the format control string are ignored by printf.
 - b) Various control characters, such as newline and tab, must be represented by escape sequences.
 - c) An escape sequence is represented by a backslash (\), followed by a particular escape character.
 - d) All of the above statements are *true*.

Answer: a) is *false*. Actually, printf displays any literal characters included in the format control string.

- **2** *(Multiple Choice)* Which escape sequence is described by "Cause an audible (bell) or visual alert (typically, flashing the window in which the program is running)"?
 - a) \b.
 - b) \r.
 - c) \a.
 - d) \v.

Answer: \a.

9.11 Formatted Input with scanf

Precise *input formatting* can be accomplished with scanf. Every scanf statement contains a format control string that describes the format of the data to be input. The

466 Chapter 9 Formatted Input/Output

format control string consists of conversion specifiers and literal characters. Function scanf has the following input formatting capabilities:

- 1. Inputting all types of data.
- 2. Inputting specific characters from an input stream.
- 3. Skipping specific characters in the input stream.

9.11.1 scanf Syntax

Function scanf is written in the following form:

scanf(format-control-string, other-arguments);

The *format-control-string* describes the input formats, and *other-arguments* are pointers to variables in which the inputs will be stored.

When inputting data, prompt the user for one data item or a few data items at a time. Avoid asking the user to enter many data items in response to a single prompt. Always consider what the user and your program will do when incorrect data is entered—for example, a value for an integer that's nonsensical in a program's context, or a string with missing punctuation or spaces.

9.11.2 scanf Conversion Specifiers

The following table summarizes the conversion specifiers used to input all types of data. Note that the d and i conversion specifiers have different meanings for input with scanf, but are interchangeable for output with printf.

Conversion specifier	Description
Integers	
d	Read an optionally signed decimal integer. The corresponding argument is a pointer to an int.
i	Read an optionally signed decimal, octal or hexadecimal integer. The corresponding argument is a pointer to an int.
0	Read an octal integer. The corresponding argument is a pointer to an unsigned int.
u	Read an unsigned decimal integer. The corresponding argument is a pointer to an unsigned int.
x or X	Read a hexadecimal integer. The corresponding argument is a pointer to an unsigned int.
h, 1 and 11	Place before any integer conversion specifier to indicate that a short, long or long long integer is to be input.
Floating-point numbers	
e, E, f, g or G	Read a floating-point value. The corresponding argument is a pointer to a floating-point variable.

Conversion specifier	Description
1 or L	Place before any floating-point conversion specifier to indicate that a double or long double value is to be input. The corresponding argument is a pointer to a double or long double variable.
Characters and strings	
С	Read a character. The corresponding argument is a pointer to a char; no null ('\0') is added.
S	Read a string. The corresponding argument is a pointer to an array of type char that's large enough to hold the string and a terminating null ('\0') character—which is automatically added.
Scan set	
[scan characters]	Scan a string for a set of characters that are stored in an array.
Miscellaneous	
р	Read an address of the same form produced when an address is output with %p in a printf statement.
n	Store the number of characters input so far in this call to scanf. The corresponding argument must be a pointer to an int.
%	Skip a percent sign (%) in the input.

9.11.3 Reading Integers

Figure 9.12 reads integers with the various integer conversion specifiers and displays the integers as decimal numbers. Conversion specification %i can input decimal, octal and hexadecimal integers.

```
// fig09_12.c
2 // Reading input with integer conversion specifiers
3 #include <stdio.h>
4
5
   int main(void) {
6
       int a = 0;
7
       int b = 0;
8
       int c = 0;
9
       int d = 0:
10
       int e = 0;
       int f = 0;
11
12
       int g = 0;
13
       puts("Enter seven integers: ");
14
15
       scanf("%d%i%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g);
16
17
       puts("\nThe input displayed as decimal integers is:");
       printf("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
18
19
    }
```

Fig. 9.12 Reading input with integer conversion specifiers. (Part 1 of 2.)

```
Enter seven integers:
-70 -70 070 0x70 70 70

The input displayed as decimal integers is:
-70 -70 56 112 56 70 112
```

Fig. 9.12 Reading input with integer conversion specifiers. (Part 2 of 2.)

9.11.4 Reading Floating-Point Numbers

When inputting floating-point numbers, any of the floating-point conversion specifiers e, E, f, g or G can be used. Figure 9.13 reads three floating-point numbers, one with each of the three types of floating conversion specifiers, and displays all three numbers with conversion specifier f.

```
I // fig09_13.c
2 // Reading input with floating-point conversion specifiers
3 #include <stdio.h>
5 int main(void) {
6
       double a = 0.0;
7
       double b = 0.0;
8
       double c = 0.0;
9
       puts("Enter three floating-point numbers:");
10
       scanf("%1e%1f%1g", &a, &b, &c);
11
12
       puts("\nUser input displayed in plain floating-point notation:");
13
14
       printf("%f\n%f\n%f\n", a, b, c);
15
   }
Enter three floating-point numbers:
1.27987 1.27987e+03 3.38476e-06
User input displayed in plain floating-point notation:
1.279870
1279.870000
0.000003
```

Fig. 9.13 Reading input with floating-point conversion specifiers.

9.11.5 Reading Characters and Strings

Characters and strings are input using the conversion specifiers c and s, respectively. Figure 9.14 prompts the user to enter a string. The program inputs the first character of the string with **%c** and stores it in the character variable x, then inputs the remainder of the string with **%s** and stores it in character array y.

```
// fig09_14.c
   // Reading characters and strings
3
  #include <stdio.h>
4
5
   int main(void) {
6
       char x = ' \setminus 0';
       char y[9] = "";
7
8
       printf("%s", "Enter a string: ");
9
       scanf("%c%8s", &x, y);
10
П
       printf("The input was '%c' and \"%s\"\n", x, y);
12
13
Enter a string: Sunday
The input was 'S' and "unday"
```

Fig. 9.14 Reading characters and strings.

9.11.6 Using Scan Sets

A sequence of characters can be input using a scan set—a set of characters enclosed in square brackets, [], and preceded by a percent sign in the format control string. A scan set scans the characters in the input stream, looking only for those characters that match characters contained in the scan set. Each time a character is matched, it's stored in the scan set's corresponding character array argument. The scan set stops inputting characters when scanf encounters a character not contained in the scan set. If the first character in the input stream does not match a character in the scan set, scanf does not modify its corresponding array argument. Figure 9.15 uses the scan set [aeiou] to scan the input stream for vowels. For our input "ooeeooahah", the first seven letters are input. The eighth letter (h) is not in the scan set, so scanf stops scanning for characters.

```
// fig09_15.c
   // Using a scan set
2
3 #include <stdio.h>
4
5
   int main(void) {
       char z[9] = "";
6
7
       printf("%s", "Enter string: ");
8
9
       scanf("%8[aeiou]", z); // search for set of characters
10
       printf("The input was \"%s\"\n", z);
11
12
    }
```

Fig. 9.15 Using a scan set. (Part 1 of 2.)

```
Enter string: ooeeooahah
The input was "ooeeooa"
```

Fig. 9.15 Using a scan set. (Part 2 of 2.)

Inverting the Scan Set

An inverted scan set can scan for characters *not* contained in the scan set. To create an inverted scan set, place a caret (^) in the square brackets before the scan characters. When a character contained in the inverted scan set is encountered, input terminates. Figure 9.16 uses the inverted scan set [^aeiou] to search for "non-vowels."

```
I // fig09_16.c
2 // Using an inverted scan set
3 #include <stdio.h>
5 int main(void) {
6
       char z[9] = "";
7
       printf("%s", "Enter a string: ");
8
9
       scanf("%8[^aeiou]", z); // inverted scan set
10
       printf("The input was \"%s\"\n", z);
11
   }
12
Enter a string: String
The input was "Str"
```

Fig. 9.16 | Using an inverted scan set.

9.11.7 Using Field Widths

A field width can be used in a scanf conversion specifier to read a specific number of characters from the input stream. Figure 9.17 inputs a series of consecutive digits as a two-digit integer and an integer consisting of the remaining digits in the input stream.

```
// fig09_17.c
// Inputting data with a field width
#include <stdio.h>

int main(void) {
   int x = 0;
   int y = 0;

printf("%s", "Enter a six digit integer: ");
scanf("%2d%d", &x, &y);
```

Fig. 9.17 | Inputting data with a field width. (Part 1 of 2.)

```
printf("The integers input were %d and %d\n", x, y);

Enter a six digit integer: 123456
The integers input were 12 and 3456
```

Fig. 9.17 | Inputting data with a field width. (Part 2 of 2.)

9.11.8 Skipping Characters in an Input Stream

You may want to skip certain characters in the input stream. Whitespace characters, such as space, newline and tab, at the beginning of a format control string skip all leading whitespace. Other literal characters ignore those characters at specific positions in the input. For example, your program might input a date as

```
11-10-1999
```

Each number in the date needs to be stored, but the dashes that separate the numbers can be discarded. To eliminate unnecessary characters, include them in scanf's format control string. For example, to discard the dashes in the input, use the statement

```
scanf("%d-%d-%d", &month, &day, &year);
```

Assignment Suppression Character

Although the preceding scanf does eliminate the dashes in the input, it's possible that the user might enter the date as

```
10/11/1999
```

In this case, the preceding scanf would not eliminate the unnecessary characters. For this reason, scanf provides the assignment suppression character *. This character enables scanf to read and discard data from the input without assigning it to a variable. Figure 9.18 uses the assignment suppression character in the %c conversion specification to indicate that a character appearing in the input stream should be read and discarded. Only the month, day and year are stored. We print the variable's values to demonstrate that they're input correctly. The argument lists for each scanf call do not contain variables for the conversion "%*c" specifiers containing the assignment suppression character. The corresponding characters are simply discarded.

```
// fig09_18.c
// Reading and discarding characters from the input stream
#include <stdio.h>

int main(void) {
   int month = 0;
   int day = 0;
   int year = 0;
```

Fig. 9.18 Reading and discarding characters from the input stream. (Part 1 of 2.)

```
printf("%s", "Enter a date in the form mm-dd-yyyy: ");
9
       scanf("%d%*c%d%*c%d", &month, &day, &year);
10
       printf("month = %d day = %d year = %d\n\n", month, day, year);
11
12
       printf("%s", "Enter a date in the form mm/dd/yyyy: ");
13
       scanf("%d%*c%d%*c%d", &month, &day, &year);
14
       printf("month = %d day = %d year = %d\n", month, day, year);
15
16
Enter a date in the form mm-dd-yyyy: 07-04-2021
month = 7 day = 4 year = 2021
Enter a date in the form mm/dd/yyyy: 01/01/2021
month = 1 day = 1 year = 2021
```

Fig. 9.18 Reading and discarding characters from the input stream. (Part 2 of 2.)

Self Check

- [(Multiple Choice) Which of the following statements is false?
 - a) A scan set scans the characters in the input stream, looking only for those characters that match characters in the scan set.
 - b) Each time a character is matched, it's stored in the scan set's corresponding character array argument.
 - c) The scan set stops inputting characters when a character that's not contained in the scan set is encountered.
 - d) If the stream's first character matches a character in the scan set, scanf does not modify the corresponding array argument.

Answer: d) is *false*. Actually, if the stream's first character *does not match* a character in the scan set, scanf does not modify the corresponding array argument.

2 (*Fill-In*) A(n) _____ can be used in a scanf conversion specifier to read a specific number of characters from the input stream.

Answer: field width.

3 (Fill-In) The scanf _____ character ____ enables scanf to read and discard data from the input without assigning it to a variable.

Answer: assignment suppression, *.

SEC ? 9.12 Secure C Programming

The C standard lists many cases in which using incorrect library-function arguments can result in undefined behaviors. These can cause security vulnerabilities, so they should be avoided. Such problems can occur when using printf (or any of its variants, such as sprintf, fprintf_s, etc.) with improperly formed conversion specifications. CERT rule FIO00-C (https://wiki.sei.cmu.edu/) discusses these issues. It presents a table showing the valid combinations of formatting flags, length modifiers and conversion-specifier characters that can be used to form conversion

specifications. The table also shows the proper argument type for each valid conversion specification. As you study *any* programming language, if the language specification says that doing something can lead to undefined behavior, avoid doing it to prevent security vulnerabilities.



Self Check

I (*TruelFalse*) Undefined behaviors can cause security vulnerabilities, so they should be avoided.

Answer: True.

Summary

Section 9.2 Streams

- Input and output are performed with streams (p. 450), which are sequences of bytes.
- The standard input stream is connected to the keyboard. The standard output and error streams are connected to the computer screen (p. 450).
- Operating systems allow the standard streams to be **redirected** to other devices.

Section 9.3 Formatting Output with printf

- A format control string (p. 451) describes the formats for values to output. It consists of conversion specifiers, flags, field widths, precisions and literal characters.
- A conversion specification (p. 451) consists of a % (p. 451) and a conversion specifier.

Section 9.4 Printing Integers

• Integers are printed with the following conversion specifiers (p. 452): **d** or **i** for optionally signed integers, **o** for unsigned integers in octal form, **u** for unsigned integers in decimal form and **x** or **X** for unsigned integers in hexadecimal form. The modifiers **h**, 1 or 11 are prefixed to the preceding conversion specifiers to indicate a short, long or long long integer.

Section 9.5 Printing Floating-Point Numbers

- Floating-point values are printed with the following conversion specifiers: **e** or **E** (p. 454) for exponential notation, **f** (p. 454) for regular floating-point notation, and **g** or **G** for either **e** (or **E**) notation or **f** notation. When the **g** (or **G**, p. 454) conversion specifier is indicated, the **e** (or **E**) conversion specifier is used if the value's exponent is less than -4 or greater than or equal to the precision with which the value is printed.
- The **precision** for the g and G conversion specifiers indicates the maximum number of significant digits printed.

Section 9.6 Printing Strings and Characters

- The conversion specifier **c** (p. 456) prints a **character**.
- The conversion specifier **s** (p. 456) prints a **string of characters** ending in the null character.

Section 9.7 Other Conversion Specifiers

- The conversion specifier **p** (p. 457) displays an **address** in an implementation-defined manner (on many systems, hexadecimal notation is used).
- The conversion specifier **%%** (p. 457) causes a literal % to be output.

Section 9.8 Printing with Field Widths and Precision

- If the **field width** (p. 451) is larger than the object being printed, the object is **right-aligned** by default.
- Field widths can be used with all conversion specifiers.
- Precision for integer conversion specifiers indicates the minimum number of digits printed.
- Precision for floating-point conversion specifiers e, E and f indicates the number of digits after the decimal point. Precision for floating-point conversion specifiers g and G indicates the number of significant digits to appear.
- Precision for conversion specifier s indicates the number of characters to print.
- The field width and the precision can be combined by placing the field width, followed by a decimal point, followed by the precision between the percent sign and the conversion specifier.
- It's possible to specify the **field width** and the **precision** through **integer expressions** in the argument list following the format control string. To do so, use an asterisk (*) for the field width or precision. The matching argument in the argument list is used in place of the asterisk.

Section 9.9 printf Format Flags

- The flag left-aligns its argument in a field.
- The + flag (p. 462) prints a plus sign for positive values and a minus sign for negative values.
- The **space flag** (p. 462) prints a space preceding a positive value that's not displayed with the + flag.
- The # flag (p. 463) prefixes 0 to octal values and 0x or 0X to hexadecimal values and forces the decimal point to be printed for floating-point values printed with e, E, f, g or G.
- The o flag (p. 463) prints leading zeros for a value that does not occupy its entire field width.

Section 9.10 Printing Literals and Escape Sequences

• Most literal characters to be printed in a printf statement can simply be included in the format control string. However, there are several "problem" characters, such as the quotation mark (", p. 465) that delimits the format control string itself. Various control characters, such as newline and tab, must be represented by escape sequences. An escape sequence is represented by a backslash (\) followed by a particular escape character.

Section 9.11 Formatted Input with scanf

- Input formatting is accomplished with the scanf library function.
- scanf inputs integers with the conversion specifiers **d** and **i** (p. 467) for optionally signed integers and **o**, **u**, **x** or **X** for unsigned integers in octal, decimal and hexadecimal formats. The modifiers **h**, **1** or **11** are placed before an integer conversion specifier to input a short, long or long long integer.
- scanf inputs floating-point values with the conversion specifiers **e**, **E**, **f**, **g** or **G**. The modifiers 1 and L are placed before any of the floating-point conversion specifiers to indicate that the input value is a double or long double value.
- scanf inputs characters with the conversion specifier **c** (p. 468).
- scanf inputs strings with the conversion specifier **s** (p. 468).
- A scanf with a scan set (p. 469) scans the characters in the input, looking only for those characters that match characters contained in the scan set. Each matching character is stored in a character array. Input stops when a character not contained in the scan set is encountered.

- To create an **inverted scan set** (p. 470), place a caret (^) in the square brackets before the scan characters. **scanf** stores characters not appearing in the inverted scan set and stops when a character contained in the inverted scan set is encountered.
- scanf inputs address values with the conversion specifier p.
- Conversion specifier **n** stores the **number of characters input** so far in the current scanf. The corresponding argument is a pointer to int.
- The assignment suppression character (*, p. 471) reads and discards data from the input stream.
- A field width is used in scanf to read a specific number of characters from the input stream.

Self-Review Exercises

hexadecimal integers.

9.1	Fill-In the blanks in each of the following:
	a) Input and output are dealt with in the form of
	b) The stream is normally connected to the keyboard.
	c) The stream is normally connected to the computer screen.
	d) Precise output formatting is accomplished with the function.
	e) The format control string may contain,, and
	·
	f) The conversion specifier or may be used to output a signed
	decimal integer.
	g) The conversion specifiers, and display unsigned inte-
	gers in octal, decimal and hexadecimal form.
	h) The modifiers and are placed before the integer conversion
	specifiers to display short or long integer values.
	i) The conversion specifier displays a floating-point value in exponen-
	tial notation.
	j) The modifier is placed before any floating-point conversion specifier
	to display a long double value.
	k) The conversion specifiers e, E and f are displayed with digits of pre-
	cision to the decimal point's right if no precision is specified.
	l) The conversion specifiers and print strings and characters.
	m) All strings end in the character.
	n) The field width and precision in a printf conversion specifier can be con-
	trolled with integer expressions by substituting a(n) for the field
	width or for the precision and placing an integer expression in the corre-
	sponding argument.
	o) The flag left-aligns output in a field.
	p) The flag displays values with either a plus sign or a minus sign.
	q) Precise input formatting is accomplished with the function.
	r) A(n) scans a string for specific characters and stores the characters in
	an array.
	s) The conversion specifier inputs optionally signed octal, decimal and

- t) The conversion specifiers _____ can be used to input a double value.
- u) The _____ reads and discards data from the input stream without assigning it to a variable.
- v) A(n) _____ can be used in a scanf conversion specifier to indicate that a specific number of characters or digits should be read from the input stream.
- **9.2** Find the error in each of the following and explain how it can be corrected.
 - a) The following statement should print the character 'c'. printf("%s\n", 'c');
 - b) The following statement should print 9.375%. printf("%.3f%", 9.375);
 - c) The following statement should print the first character of "Monday".

 printf("%c\n", "Monday");
 - d) puts(""A string in quotes"");
 - e) printf(%d%d, 12, 20);
 - f) printf("%c", "x");
 - g) printf("%s\n", 'Richard');
- **9.3** Write a statement for each of the following:
 - a) Print 1234 right-aligned in a 10-digit field.
 - b) Print 123.456789 in exponential notation with a sign (+ or -) and 3 digits of precision.
 - c) Read a double value into variable number.
 - d) Print 100 in octal form preceded by 0.
 - e) Read a string into character array string.
 - f) Read characters into array n until a nondigit character is encountered.
 - g) Use integer variables x and y to specify the field width and precision used to display the double value 87.4573.
 - h) Read a value of the form 3.5%. Store the percentage in float variable percent and eliminate the % from the input stream. Do not use the assignment suppression character.
 - i) Print 3.333333 as a long double value with a sign (+ or -) in a field of 20 characters with a precision of 3.

Answers to Self-Review Exercises

- **9.1** a) streams. b) standard input. c) standard output. d) printf. e) conversion specifiers, flags, field widths, precisions, literal characters. f) d, i. g) o, u, x (or X). h) h, 1. i) e (or E). j) L. k) 6. l) s, c. m) NULL ('\0'). n) asterisk (*). o) (minus). p) + (plus). q) scanf. r) scan set. s) i. t) 1e, 1E, 1f, 1g or 1G. u) assignment suppression character (*). v) field width.
- **9.2** See the answers below:
 - a) Error: Conversion specifier s expects an argument of type pointer to char. Correction: To print the character 'c', use the conversion specifier %c or change 'c' to "c".

- b) Error: Trying to print the literal character % without using the conversion specifier %%.
 - Correction: Use %% to print a literal % character.
- c) Error: Conversion specifier c expects an argument of type char.

 Correction: To print the first character of "Monday" use the conversion specifier %.1s.
- d) Error: Trying to print the literal character " without using the \" escape sequence.
 - Correction: Replace each quote in the inner set of quotes with \".
- e) Error: The format control string is not enclosed in double quotes. Correction: Enclose %d%d in double quotes.
- f) Error: The character x is enclosed in double quotes. Correction: Character constants to be printed with %c must be enclosed in single quotes.
- g) Error: The string to be printed is enclosed in single quotes.
 Correction: Use double quotes instead of single quotes to represent a string.
- 9.3 a) printf("%10d\n", 1234);
 b) printf("%+.3e\n", 123.456789);
 c) scanf("%1f", &number);
 d) printf("%#o\n", 100);
 e) scanf("%s", string);
 f) scanf("%[0123456789]", n);
 g) printf("%*.*f\n", x, y, 87.4573);
 h) scanf("%f%%", &percent);
 i) printf("%+20.3Lf\n", 3.333333);

Exercises

- **9.4** Write a printf or scanf statement for each of the following:
 - a) Print unsigned integer 40000 left-aligned in a 15-digit field with 8 digits.
 - b) Read a hexadecimal value into variable hex.
 - c) Print 200 with and without a sign.
 - d) Print 100 in hexadecimal form preceded by 0x.
 - e) Read characters into array s until the letter p is encountered.
 - f) Print 1.234 in a 9-digit field with preceding zeros.
 - g) Read a time of the form hh:mm:ss, storing the parts of the time in the integer variables hour, minute and second. Skip the colons (:) in the input stream. Use the assignment suppression character.
 - h) Read a string of the form "characters" from the standard input. Store the string in character array s. Eliminate the quotation marks from the input.
 - i) Read a time of the form hh:mm:ss, storing the parts of the time in the integer variables hour, minute and second. Skip the colons (:) in the input stream. Do not use the assignment suppression character.

9.5 Show what each of the following statements prints. If a statement is incorrect, indicate why.

```
a) printf("%-10d\n", 10000);
b) printf("%c\n", "This is a string");
c) printf("%*.*lf\n", 8, 3, 1024.987654);
d) printf("%#o\n%#X\n%#e\n", 17, 17, 1008.83689);
e) printf("% ld\n%+ld\n", 10000000, 10000000);
f) printf("%10.2E\n", 444.93738);
g) printf("%10.2g\n", 444.93738);
h) printf("%d\n", 10.987);
```

9.6 Find the error(s) in each of the following program segments. Explain how each error can be corrected.

```
a) printf("%s\n", 'Happy Birthday');
b) printf("%c\n", 'Hello');
c) printf("%c\n", "This is a string");
d) The following statement should print "Bon Voyage":
    printf(""%s"", "Bon Voyage");
e) char day[] = "Sunday";
    printf("%s\n", day[3]);
f) puts('Enter your name: ');
g) printf(%f, 123.456);
h) The following statement should print the characters 'O' and 'K':
    printf("%s%s\n", 'O', 'K');
i) char s[10];
    scanf("%c", s[7]);
```

9.7 (*Differences Between %d and %i*) Write a program to test the difference between the %d and %i conversion specifiers when used in scanf statements. Ask the user to enter two integers separated by a space. Use the statements

```
scanf("%i%d", &x, &y);
printf("%d %d\n", x, y);
```

to input and print the values. Test the program with the following sets of input data:

```
10 10
-10 -10
010 010
0x10 0x10
```

- **9.8** (*Printing Numbers in Various Field Widths*) Write a program that prints the integer value 12345 and the floating-point value 1.2345 in fields of various sizes. What happens when the values are printed in fields containing fewer digits than the values?
- **9.9** (*Rounding Floating-Point Numbers*) Write a program that prints 100.453627 rounded to the nearest digit, tenth, hundredth, thousandth and ten-thousandth.

9.10 (*Temperature Conversions*) Write a program that converts integer Fahrenheit temperatures from 0 to 212 degrees to floating-point Celsius temperatures with 3 digits of precision. Perform the calculation using the formula

```
celsius = 5.0 / 9.0 * (fahrenheit - 32);
```

Display the output in two right-aligned columns of 10 characters each. Precede the Celsius temperatures by a sign for both positive and negative values.

- **9.11** (Escape Sequences) Write a program to test the escape sequences \', \", \?, \\, \a, \b, \n, \r and \t. For the escape sequences that move the cursor, print a character before and after printing the escape sequence so it's clear where the cursor has moved.
- **9.12** (*Printing a Question Mark*) Write a program that determines whether? can be printed as part of a printf format control string as a literal character rather than using the \? escape sequence.
- **9.13** (Reading an Integer with Each scanf Conversion Specifier) Write a program that inputs the value 437 using each of the scanf integer conversion specifiers. Print each input value using all the integer conversion specifiers.
- **9.14** (Outputting a Number with the Floating-Point Conversion Specifiers) Write a program that uses each of the conversion specifiers e, f and g to input the value 1.2345. Print the values of each variable to prove that each conversion specifier can be used to input this same value.
- **9.15** (*Reading Strings in Quotes*) In some programming languages, strings are entered surrounded by either single or double quotation marks. Write a program that reads the three strings suzy, "suzy" and 'suzy'. Are the single and double quotes ignored by C or read as part of the string?
- **9.16** (*Printing a Question Mark as a Character Constant*) Write a program that determines whether? can be printed as the character constant '?' rather than the character constant escape sequence '\?'. Use the conversion specifier %c in the format control string of a printf statement.
- **9.17** (Using %g with Various Precisions) Write a program that uses the conversion specifier g to output the value 9876.12345. Print the value with precisions ranging from 1 to 9.

Structures, Unions, Bit Manipulation and Enumerations



10

Objectives

In this chapter, you'll:

- Create and use structs, unions and enums.
- Understand self-referential structs.
- Learn about the operations that can be performed on struct instances.
- Initialize struct members.
- Access struct members.
- Pass struct instances to functions by value and by reference.
- Use typedefs to create aliases for existing type names.
- Learn the operations that can be performed on **union**s.
- Initialize unions.
- Manipulate integer data with the bitwise operators.
- Create bit fields for storing data compactly.
- Use enum constants.
- Consider the security issues of working with structs, bit manipulation and enums.

- 10.1 Introduction
- **10.2** Structure Definitions
 - 10.2.1 Self-Referential Structures
 - 10.2.2 Defining Variables of Structure Types
 - 10.2.3 Structure Tag Names
 - 10.2.4 Operations That Can Be Performed on Structures
- **10.3** Initializing Structures
- 10.4 Accessing Structure Members withand ->
- **10.5** Using Structures with Functions
- 10.6 typedef
- 10.7 Random-Number Simulation Case Study: High-Performance Card Shuffling and Dealing
- 10.8 Unions
 - 10.8.1 Union Declarations
 - 10.8.2 Allowed union Operations

- 10.8.3 Initializing **union**s in Declarations
- 10.8.4 Demonstrating **union**s
- **10.9** Bitwise Operators
 - 10.9.1 Displaying an Unsigned Integer's Bits
 - 10.9.2 Making Function displayBits
 More Generic and Portable
 - 10.9.3 Using the Bitwise AND, Inclusive OR, Exclusive OR and Complement Operators
 - 10.9.4 Using the Bitwise Left- and Right-Shift Operators
 - 10.9.5 Bitwise Assignment Operators
- **10.10** Bit Fields
 - 10.10.1 Defining Bit Fields
 - 10.10.2 Using Bit Fields to Represent a Card's Face, Suit and Color
 - 10.10.3 Unnamed Bit Fields
- **10.11** Enumeration Constants
- 10.12 Anonymous Structures and Unions
- **10.13** Secure C Programming

Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises |
Special Section: Raylib Game-Programming Case Studies

10.1 Introduction

Structures are collections of related variables under one name, known as aggregates in the C standard. Structures may contain many variables of different types. That's in contrast to arrays, which contain only elements of the same type. Here, we'll discuss:

- typedefs—for creating *aliases* for previously defined data types.
- unions—similar to structures, but with members that *share* the *same* storage.
- bitwise operators—for manipulating the bits of integral operands.
- bit fields—unsigned int or int members of structures or unions for which you specify the number of bits in which the members are stored, helping you pack information tightly.
- enumerations—sets of integer constants represented by identifiers.

In Chapters 11 and 12, you'll see that

- structures commonly define records to be stored in files, and
- pointers and structures facilitate forming data structures such as linked lists, queues, stacks and trees.

✓ Self Check

I (*Fill-In*) _____ are similar to structures, but with members that share the same storage space.

Answer: unions.

2 (*Fill-In*) _____ are sets of integer constants represented by identifiers. **Answer:** Enumerations.

10.2 Structure Definitions

Structures are derived data types—they're constructed using objects of other types. The keyword **struct** introduces a structure definition, as in

```
struct card {
   const char *face:
   const char *suit;
};
```

The identifier card is the structure tag, which you use with struct to declare variables of the structure type—e.g., struct card. Variables declared within a struct's braces are the structure's members. A struct's members must have unique names, though separate structure types may contain members of the same name without conflict. Each structure definition ends with a semicolon.

The struct card definition contains const char * members face and suit. Structure members can be const or non-const primitive-type variables (e.g., ints, doubles, etc.) or aggregates, such as arrays or other struct-type objects. Chapter 6 showed that an array's elements all have the same type. Structure members, however, can be of different types. For example, the following struct contains char array members for an employee's first and last names, an int member for the employee's age and a double member for the employee's hourly salary:

```
struct employee {
   char firstName[20];
   char lastName[20];
   int age;
   double hourlySalary;
}:
```

10.2.1 Self-Referential Structures

A struct type may not contain a variable of its own struct type (which is a compi- 💢 ERR lation error), but it may contain a pointer to that struct type. For example, the updated struct employee below contains a pointer to the employee's manager, which would be another struct employee object:



```
struct employee {
   char firstName[20];
   char lastName[20]:
   unsigned int age;
   double hourlySalary;
   struct employee *managerPtr; // pointer
```

A structure containing a member that's a pointer to the *same* struct type is a selfreferential structure. Self-referential structures are used to build linked data structures.

10.2.2 Defining Variables of Structure Types

A structure definition does not reserve any space in memory. Rather, it creates a new data type you can use to define variables. It's like a blueprint showing how to build instances of that struct. The following statements reserve memory for variables using the type struct card:

```
struct card myCard;
struct card deck[52];
struct card *cardPtr;
```

Variable myCard is a struct card object, array deck consists of 52 struct card objects, and cardPtr is a pointer to a struct card object.

Variables of a given structure type may also be defined by placing a comma-separated list of variable names between the struct's closing brace and terminating semicolon. For example, you can incorporate the preceding definitions into the struct card definition:

```
struct card {
   const char *face;
   const char *suit;
} myCard, deck[52], *cardPtr;
```

10.2.3 Structure Tag Names

The structure tag name is optional. If a structure definition does not specify a tag name, you must define any variables of the type, as shown in the preceding code snippet. Always provide a structure tag name so you can declare new variables of that type later.

10.2.4 Operations That Can Be Performed on Structures

You can perform the following operations on structs:

- assigning one struct variable to another of the *same* type (Section 10.7)—for a pointer member, this copies only the address stored in the pointer,
- taking the address (&) of a struct variable (Section 10.4),
- accessing a struct variable's members (Section 10.4),
- using the sizeof operator to determine a struct variable's size, and
- zero initializing a struct variable in its definition, as in

```
struct card myCard = {};
```

ERR (X) Assigning a structure of one type to one of a different type is a compilation error.

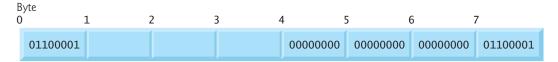
Comparing Structure Objects Is Not Allowed

Structures may *not* be compared using operators == and !=, because structure members may not be stored in consecutive bytes of memory. Sometimes there are "holes" in a structure because computers store some data types only on certain memory boundaries, such as half-word, word or double-word boundaries. This is machine-dependent. A word is a memory unit used to store data in a computer, usually four bytes or eight bytes.

Consider the following structure definition, which also defines variables sample1 and sample2:

```
struct example {
   char c:
   int i;
} sample1, sample2;
```

A computer with four-byte words might require that each struct example member be aligned on a word boundary, i.e., at the beginning of a word. The following diagram shows a possible memory alignment for a struct example variable that has been assigned the character and the integer 97. We show the bit representations here.



If each member is stored beginning at a word boundary, each struct example variable has a three-byte hole in bytes 1–3. The hole's value is *unspecified*. Even if sample1's and sample2's member values are equal, the holes are not likely to contain identical values, so the structures are not necessarily equal. Data type sizes and memory alignment considerations are machine-dependent.



Self Check

I (Multiple Choice) Consider the struct name definition:

```
struct name {
   const char *first;
   const char *last;
};
```

Which of the following statements a), b) or c) is *false?*

- a) Keyword struct introduces the structure definition.
- b) The structure tag name can be used with struct to declare variables of the structure type.
- c) Variables declared within a struct's braces are the structure's members, which must have unique names.
- d) All of the above statements are *true*.

Answer: d.

- (Multiple Choice) Which of the following a), b) or c) is not a valid operation that may be performed on a structure?
 - a) Assigning struct variables to struct variables of the same type.
 - b) Dereferencing a struct variable.
 - c) Accessing a struct variable's members and using sizeof to determine the size of a struct variable.
 - d) All of the above statements are true.

Answer: b) is not valid. You cannot dereference a struct because it's not a pointer, but you can take a struct's address with &.

10.3 Initializing Structures

Like arrays, you can initialize a struct variable via an initializer list. For example, the following statement creates variable myCard using type struct card (Section 10.2) and initializes member face to "Three" and member suit to "Hearts":

```
struct card myCard = {"Three", "Hearts"};
```

If there are fewer initializers than members, the remaining members are automatically initialized to 0 or NULL (for pointer members). Structure variables defined outside a function definition (i.e., externally) are initialized to 0 or NULL if they're not explicitly initialized in the external definition. You may also assign structure variables to other structure variables of the same type or assign values to individual structure members.

Self Check

[(True/False) If there are fewer initializers in the list than members in the structure, the remaining members are not initialized.

Answer: *False.* Actually, they're initialized to 0 (or NULL if the member is a pointer).

2 (*True*/*False*) You may assign structure variables to other structure variables of the same type or assign values to individual structure members.

Answer: True.

10.4 Accessing Structure Members with . and ->

You can access structure members with:

- the structure member operator (.), or dot operator, and
- the structure pointer operator (->), or arrow operator.

Structure Member Operator (.)

The structure member operator accesses a structure member via a structure variable name. For example, using the structure variable myCard from Section 10.3, we can print the suit member with the statement:

```
printf("%s", myCard.suit); // displays Hearts
```

Structure Pointer Operator (->)

You can access a structure member via a pointer to the structure using the structure pointer operator—a minus (-) sign and a greater than (>) sign with no intervening spaces. If the pointer cardPtr points to the struct card object myCard we defined earlier, we can print its member suit with the statement:

```
printf("%s", cardPtr->suit); // displays Hearts
```

The expression cardPtr->suit is equivalent to (*cardPtr).suit, which dereferences the pointer and accesses the member suit using the structure member operator (.). The parentheses are needed here because the structure member operator (.) has higher precedence than the pointer dereferencing operator (*). The structure pointer operator and

structure member operator have the highest precedence and group from left-to-right, along with parentheses (for calling functions) and brackets ([]) used for array indexing.

Spacing Conventions

Do not put spaces around the -> and . (dot) operators to emphasize that the expressions the operators are contained in are essentially single variable names. Inserting (X) ERR space between the structure pointer operator's - and > or between any other multiplekeystroke operator's components (except ?:) is a syntax error.



Demonstrating the Structure Member and Structure Pointer Operators

Figure 10.1 refers to members of structure myCard using the structure member and structure pointer operators. Lines 16-17 assign "Ace" and "Spades" to myCard's members. Line 19 assigns myCard's address to cardPtr. Lines 21–23 display myCard's members using:

- the structure member operator and variable name myCard,
- the structure pointer operator and pointer cardPtr, and
- the structure member operator with dereferenced pointer cardPtr.

```
I // fig10_01.c
 2 // Structure member operator and
 3 // structure pointer operator
 4 #include <stdio.h>
 5
 6 // card structure definition
 7
    struct card {
       const char *face; // define pointer face
       const char *suit; // define pointer suit
 9
10
    };
11
12 int main(void) {
       struct card myCard; // define one struct card variable
13
14
       // place strings into myCard
15
       myCard.face = "Ace";
16
       myCard.suit = "Spades";
17
18
19
       struct card *cardPtr = &myCard; // assign myCard
20
       printf("%s of %s\n", myCard.face, myCard.suit);
21
       printf("%s of %s\n", cardPtr->face, cardPtr->suit);
22
23
       printf("%s of %s\n", (*cardPtr).face, (*cardPtr).suit);
24
   }
Ace of Spades
Ace of Spades
Ace of Spades
```

Structure member operator and structure pointer operator. Fig. 10.1



Self Check

- (Fill-In) The structure member operator _____ and the structure pointer operator _____ can be used to access structure members. **Answer:** . (dot) , ->.
- 2 (True/False) The expression cardPtr->suit is equivalent to (*cardPtr).suit, which dereferences the pointer and accesses the member suit using the structure member operator. The parentheses are optional.

Answer: False. The expressions, as shown, are equivalent. The parentheses are needed here because the structure member operator (.) has higher precedence than the pointer dereferencing operator (*).

10.5 Using Structures with Functions

With structures, you can pass to functions:

- individual structure members,
 - entire structure objects, or
 - pointers to structure objects.

Individual structure members and entire structure objects are passed by value, so functions cannot modify them in the caller. To pass a structure by reference, use the PERF structure object's address. Passing structures by reference is more efficient than passing structures by value, which requires the entire structure to be copied. Arrays of structure objects—like all other arrays—are automatically passed by reference.

Passing an Array By Value

In Chapter 6, we stated that you can use a structure to pass an array by value. To do so, create a structure with an array member. Structures are passed by value, so its members are passed by value.



Self Check

[(Fill-In) Structure objects and individual structure members are passed to functions by _____. Answer: value.

2 (*Discussion*) How can you pass an array by value?

Answer: Simply place the array in a structure and pass the structure. Structures normally pass by value.

10.6 typedef

The keyword typedef enables you to create synonyms (or aliases) for previously defined types. It's commonly used to create shorter names for struct types and simplify declarations of types like function pointers. For example, the following typedef defines Card as a synonym for type struct card:

typedef struct card Card;

By convention, capitalize the first letter of typedef names to emphasize that they're synonyms for other type names.

You can now use Card to declare variables of type struct card. The declaration Card deck[52];

declares an array of 52 Card structures (i.e., variables of type struct card). Creating a new name with typedef does *not* create a new type; typedef creates an alternate *type name*, which may be used as an *alias* for an existing type name. A meaningful name helps make the program self-documenting. For example, when we read the previous declaration, we know "deck is an array of 52 Cards."

Combining typedef with struct Definitions

Programmers often use typedef to define a structure type, so a structure tag is not required. For example, the following definition also creates the structure type Card:

```
typedef struct {
   const char *face;
   const char *suit:
} Card;
```

Synonyms for Built-In Types

Using typedefs can help make a program more readable and maintainable. Often, typedef is used to create synonyms for built-in types. For example, a program requiring four-byte integers may use type int on one system and type long on another. Programs designed for portability often use typedef to create an alias for four-byte integers, such as Integer. To port the program to another platform, you can simply change the Integer typedef and recompile the program.

Self Check

(Code) Write a typedef statement that creates for structure type struct dice the shorter type name Dice.

Answer: typedef struct dice Dice;

2 (*True/False*) Creating a new name with typedef creates a new type. **Answer:** False. Creating a new name with typedef does not create a new type. It simply creates a new *type name*, which may be used as an *alias* for an existing type name.

10.7 Random-Number Simulation Case Study: High-Performance Card Shuffling and Dealing

Figure 10.2 is based on Chapter 7's card shuffling and dealing simulation. This program represents the deck of cards as an array of Card structs and uses high-performance shuffling and dealing algorithms.

```
// fig10_02.c
   // Card shuffling and dealing program using structures
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5
   #include <time.h>
 6
 7
   #define CARDS 52
 8
   #define FACES 13
 9
10 // card structure definition
11
   struct card {
       const char *face; // define pointer face
12
13
       const char *suit; // define pointer suit
14
   };
15
   typedef struct card Card; // new type name for struct card
16
17
18
   // prototypes
19 void fillDeck(Card * const deck, const char *faces[], const char *suits[]);
   void shuffle(Card * const deck);
20
21
   void deal(const Card * const deck);
22
23
   int main(void) {
       Card deck[CARDS]; // define array of Cards
24
25
26
       // initialize faces array of pointers
       const char *faces[] = { "Ace", "Deuce", "Three", "Four", "Five",
27
          "Six", "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King"};
28
29
30
       // initialize suits array of pointers
       const char *suits[] = { "Hearts", "Diamonds", "Clubs", "Spades"};
31
32
33
       srand(time(NULL)); // randomize
34
35
       fillDeck(deck, faces, suits); // load the deck with Cards
       shuffle(deck); // put Cards in random order
36
       deal(deck); // deal all 52 Cards
37
38
   }
39
40
   // place strings into Card structures
41
   void fillDeck(Card * const deck, const char * faces[],
42
       const char * suits[]) {
43
       // loop through deck
       for (size_t i = 0; i < CARDS; ++i) {
44
45
          deck[i].face = faces[i % FACES];
46
          deck[i].suit = suits[i / FACES];
47
       }
    }
48
49
```

Fig. 10.2 | Card shuffling and dealing program using structures. (Part 1 of 2.)

```
50
    // shuffle cards
    void shuffle(Card * const deck) {
51
52
       // loop through deck randomly swapping Cards
53
       for (size_t i = 0; i < CARDS; ++i) {
54
           size_t j = rand() % CARDS;
55
          Card temp = deck[i];
56
          deck[i] = deck[j];
57
          deck[j] = temp;
58
       }
59
    }
60
61
    // deal cards
    void deal(const Card * const deck) {
62
63
       // loop through deck
64
       for (size_t i = 0; i < CARDS; ++i) {
          printf("%5s of %-8s%s", deck[i].face, deck[i].suit,
      (i + 1) % 4 ? " " : "\n");
65
66
67
       }
68
    }
Three of Hearts
                     Jack of Clubs
                                        Three of Spades
                                                               Six of Diamonds
                    Eight of Spades
 Five of Hearts
                                         Three of Clubs
                                                             Deuce of Spades
 Jack of Spades
                     Four of Hearts
                                         Deuce of Hearts
                                                               Six of Clubs
Queen of Clubs
                    Three of Diamonds
                                        Eight of Diamonds
                                                              King of Clubs
                                                             Seven of Clubs
 King of Hearts
                    Eight of Hearts
                                         Queen of Hearts
Seven of Diamonds
                     Nine of Spades
                                          Five of Clubs
                                                             Eight of Clubs
  Six of Hearts
                    Deuce of Diamonds
                                          Five of Spades
                                                              Four of Clubs
Deuce of Clubs
                     Nine of Hearts
                                         Seven of Hearts
                                                              Four of Spades
  Ten of Spades
                     King of Diamonds
                                          Ten of Hearts
                                                              Jack of Diamonds
 Four of Diamonds
                      Six of Spades
                                          Five of Diamonds
                                                               Ace of Diamonds
  Ace of Clubs
                     Jack of Hearts
                                          Ten of Clubs
                                                             Queen of Diamonds
  Ace of Hearts
                      Ten of Diamonds
                                          Nine of Clubs
                                                              King of Spades
  Ace of Spades
                     Nine of Diamonds
                                        Seven of Spades
                                                             Queen of Spades
```

Fig. 10.2 Card shuffling and dealing program using structures. (Part 2 of 2.)

Line 35 calls function fillDeck (lines 41–48) to initialize the Card array in order with "Ace" through "King" of each suit. Line 36 passes the Card array to function shuffle (lines 51–59), which implements the high-performance shuffling algorithm. Function shuffle takes an array of 52 Cards as an argument. The function loops through the 52 Cards. For each Card, the algorithm chooses a random number between 0 and 51, then swaps the current Card and the randomly selected Card. The algorithm performs 52 swaps in a single pass of the entire array, and the array of Cards is shuffled! This algorithm cannot suffer from indefinite postponement like Chapter 7's shuffling algorithm. The Cards were swapped in place in the array, so the high-performance dealing algorithm in function dea1 (lines 62–68) can deal the shuffled Cards in only *one* pass of the array.

Related Exercise—Fisher-Yates Shuffling Algorithm

It's recommended that you use an unbiased shuffling algorithm for real card games. Such an algorithm ensures that all possible shuffled card sequences are equally likely to occur. Exercise 10.18 asks you to research the popular unbiased Fisher-Yates shuffling algorithm and use it to reimplement function shuffle in Fig. 10.2.



Self Check

(Code) Rewrite the following code to avoid the separate typedef statement:

```
struct name {
    const char *first;
    const char *last;
};

typedef struct name Name;

Answer:

typedef struct name {
```

const char *first;
const char *last;
} Name;

2 (*Code*) Correct the following code, which is supposed to swap elements i and j of the deck array of Cards:

```
deck[i] = deck[j];
deck[j] = deck[i];
```

Answer:

```
Card temp = deck[i];
deck[j] = deck[i];
deck[i] = temp;
```

3 (*Discussion*) Why did the card shuffling algorithm we presented in Chapter 7 suffer from indefinite postponement? Why doesn't this chapter's card shuffling algorithm suffer from indefinite postponement?

Answer: Chapter 7's card shuffling and dealing example used a 4-by-13 array to represent the four suits and 13 faces in a deck. The shuffling algorithm used sentinel-controlled looping to place 1–52 (representing dealing order) into randomly selected rows and columns. This loop could execute indefinitely if the randomly selected cells already contain one of these values. This chapter's shuffling algorithm uses countercontrolled iteration to make one pass of a one-dimensional array. The loop iterates once for each card, then terminates, so it cannot suffer from indefinite postponement.

10.8 Unions

Like a structure, a union is a derived data type, but its members share the same memory. At different times during program execution, some variables may not be relevant when others are. So, a union shares the space rather than wasting storage on variables that are not in use. A union's members can be of any type. The number of bytes used to store a union must be at least enough to hold its largest member.

In most cases, unions contain two or more items of different types. You can reference only one member—and thus only one type—at a time. It's your responsibility to reference the data with the proper type. Referencing the currently stored data with a variable of the wrong type is a logic error—the result is implementation-dependent. (X) ERR



Union Portability

The amount of memory required to store a union is implementation-dependent. A SE Operator sizeof will always return a value at least as large as the size in bytes of the union's largest member. Some unions may not port easily among computer systems. Whether a union is portable or not often depends on the memory alignment require- 🏻 📥 SE ments for a union's member types on a given system.



10.8.1 union Declarations

The following union has two members—int x and double y:

```
union number {
   int x;
   double y;
};
```

The union definition is normally placed in a header and included in all source files that use the union type. As with a struct definition, a union definition simply creates 🔺 SE a new type. It does not reserve any memory until you use the type to create variables.



10.8.2 Allowed unions Operations

The operations that can be performed on a union are:

- assigning a union to another union of the same type,
- taking a union variable's address (&),
- accessing union members via the structure member operator (.) and the structure pointer operator (->), and
- zero-initializing the union.

Two unions may not be compared using operators == and != for the same reasons that two structures cannot be compared.

10.8.3 Initializing unions in Declarations

You can initialize a union in a declaration with a value of the union's first member type. The union number in Section 10.8.1 has an int as its first member, so we can initialize an object of this type with the following statement:

```
union number value = {10};
```

If you initialize the object with a double, as in

```
union number value = {1.43};
```

C will truncate the initializer value's floating-point part—some compilers will issue a warning about this.

10.8.4 Demonstrating unions

Figure 10.3 displays a union number variable named value (line 12) as both an int and a double. This program's output is implementation-dependent.

```
I // fig10_03.c
 2 // Displaying the value of a union in both member data types
 3 #include <stdio.h>
 4
 5 // number union definition
 6 union number {
 7
       int x:
       double y;
 8
 9
   };
10
  int main(void) {
П
       union number value; // define a union variable
12
13
       value.x = 100; // put an int into the union
14
15
       puts("Put 100 in the int member and print both members:");
16
       printf("int: %d\ndouble: %.2f\n\n", value.x, value.y);
17
       value.y = 100.0; // put a double into the same union
18
19
       puts("Put 100.0 in the double member and print both members:");
20
       printf("int: %d\ndouble: %.2f\n\n", value.x, value.y);
21 }
```

Microsoft Visual Studio

```
Put 100 in the int member and print both members: int: 100 double: -92559592117433135502616407313071917486139351398276445610442752.00

Put 100.0 in the double member and print both members: int: 0 double: 100.00
```

GNU GCC and Apple Xcode

```
Put 100 in the int member and print both members:
int: 100
double: 0.00

Put 100.0 in the double member and print both members:
int: 0
double: 100.00
```

Fig. 10.3 Displaying the value of a **union** in both member data types.

Self Check

(*Discussion*) Like a struct, a union is a derived data type. How is a union different from a struct?

Answer: A union shares its memory among all of its members. Only one member may be stored in a union at any time. You must keep track of which member is currently stored.

2 (True/False) The following union definition indicates that number is a union type with members int x and double y:

```
union number {
   int x;
   double y;
};
```

For a machine with four-byte ints and eight-byte doubles, the compiler must reserve at least 12 bytes for a variable of this union type.

Answer: False. Only one of these members is active at a time, so the *union* needs to reserve only as much storage as is needed for the largest member—in this case, eight bytes.

10.9 Bitwise Operators

Computers represent all data internally as sequences of bits. Each bit can assume the value 0 or the value 1. On most systems, a sequence of eight bits forms a byte—the typical storage unit for a char variable. The bitwise operators are used to manipulate the bits of integral operands, both signed and unsigned, though unsigned integers are typically used. Bitwise data manipulations are machine-dependent. The following 📥 SE table summarizes the bitwise operators.

Operator		Description
&	bitwise AND	Compares its two operands bit by bit. The bits in the result are set to 1 if the corresponding bits in the two operands are both 1.
1	bitwise inclusive OR	Compares its two operands bit by bit. The bits in the result are set to 1 if <i>at least one</i> of the corresponding bits in the two operands is 1.
٨	bitwise exclusive OR (also known as bit- wise XOR)	Compares its two operands bit by bit. The bits in the result are set to 1 if the corresponding bits in the two operands are different.
<<	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from the right with 0 bits.
>>	right shift	Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine-dependent when the left operand is negative.
~	complement	All 0 bits are set to 1, and all 1 bits are set to 0. This is often called toggling the bits.

Detailed discussions of each bitwise operator appear in the examples that follow. The examples show the binary representations of the integer operands.

10.9.1 Displaying an Unsigned Integer's Bits

When using the bitwise operators, it's useful to display values in binary¹ to show each operator's precise effects. Figure 10.4 prints an unsigned int in its binary representation using eight-bit groups for readability. All the compilers we used to test these examples store unsigned ints in 4 bytes (32 bits) of memory.

```
I // fig10_04.c
   // Displaying an unsigned int in bits
 2
 3 #include <stdio.h>
 5
   void displayBits(unsigned int value); // prototype
 6
 7
    int main(void) {
 8
       unsigned int x = 0; // variable to hold user input
 9
10
       printf("%s", "Enter a nonnegative int: ");
       scanf("%u", &x);
11
       displayBits(x);
12
13
    }
14
    // display bits of an unsigned int value
15
    void displayBits(unsigned int value) {
16
       // define displayMask and left shift 31 bits
17
       unsigned int displayMask = 1 << 31;</pre>
18
19
       printf("%10u = ", value);
20
21
22
       // loop through bits
       for (unsigned int c = 1; c \le 32; ++c) {
23
24
          putchar(value & displayMask ? : );
25
          value <<= 1; // shift value left by 1</pre>
26
          if (c \% 8 == 0) \{ // \text{ output space after } 8 \text{ bits} 
27
28
              putchar();
29
           }
30
       }
31
32
       putchar();
33
    }
Enter a nonnegative int: 65000
     65000 = 00000000 \ 00000000 \ 111111101 \ 11101000
```

Fig. 10.4 Displaying an unsigned int in bits.

^{1.} See online Appendix E for a detailed explanation of the binary (base-2) number system.

Displaying the Bits of an Integer

Function displayBits (lines 16–33) uses the bitwise AND operator to combine variable value with the variable displayMask (line 24). Often, the bitwise AND operator is used with an operand called a mask—an integer value with specific bits set to 1. Masks are used to hide some bits in a value while selecting other bits. In function displayBits, line 18 assigns the mask variable displayMask the value

```
(10000000 00000000 00000000 00000000)
1 << 31
```

The left-shift operator shifts the value 1 from the low-order (rightmost) bit to the high-order (leftmost) bit in displayMask and fills in 0 bits from the right. Line 24

```
putchar(value & displayMask ? : );
```

determines whether to display a 1 or a 0 for the current leftmost bit of value. Combining value and displayMask with & "masks off" (hides) all the bits except the high-order bit in value—any bit "ANDed" with 0 yields 0. If the leftmost bit is 1, value & displayMask evaluates to a nonzero (true) value and line 24 displays 1; otherwise, it displays 0. Line 25 left shifts the variable value one bit with the expression value <<= 1. Function displayBits repeats these steps for each bit in value. Using the logical AND operator (&&) for the bitwise AND operator (&)—and vice versa—is a logic error. The table below summarizes the results of combining two bits with the bitwise AND operator.



Bit I	Bit 2	Bit I & Bit 2
0	0	0
0	1	0
1	0	0
1	1	1

10.9.2 Making Function displayBits More Generic and Portable

In line 18 of Fig. 10.4, we hard-coded the integer 31 to indicate that the value 1 should be shifted to the leftmost bit in the variable displayMask. Similarly, in line 23, we hardcoded the integer 32 to indicate that the loop should iterate 32 times, once for each bit in value. We assumed that unsigned into are always stored in 32 bits (four bytes) of memory. Today's popular computers generally use 32-bit- or 64-bit-word hardware architectures. As a C programmer, you'll tend to work across many hardware architectures, and sometimes unsigned ints will be stored in smaller or larger numbers of bits.

Figure 10.4 can be made more generic and portable by replacing the integers 31 📥 SE (line 18) and 32 (line 23) with expressions that calculate these integers, based on the size of an unsigned int for a given platform. The symbolic constant CHAR_BIT (defined in <1imits.h>) represents the number of bits in a byte (normally 8). Recall sizeof determines the number of bytes used to store an object or type. The expression sizeof(unsigned int) evaluates to 4 for 32-bit unsigned ints and 8 for 64-bit unsigned ints. You can replace 31 with

CHAR_BIT * sizeof(unsigned int) - 1

```
and replace 32 with
        CHAR_BIT * sizeof(unsigned int)
```

For 32-bit unsigned ints, these expressions evaluate to 31 and 32. For 64-bit unsigned ints, they evaluate to 63 and 64.

10.9.3 Using the Bitwise AND, Inclusive OR, Exclusive OR and Complement Operators

Figure 10.5 demonstrates the bitwise AND, the bitwise inclusive OR, the bitwise exclusive OR and the bitwise complement operators. The program uses function displayBits (lines 45–62) to display the unsigned int values.

```
I // fig10_05.c
 2 // Using the bitwise AND, bitwise inclusive OR, bitwise
 3 // exclusive OR and bitwise complement operators
 4 #include <stdio.h>
 5
 6 void displayBits(unsigned int value); // prototype
 7
 8 int main(void) {
       // demonstrate bitwise AND (&)
 9
       unsigned int number1 = 65535;
10
       unsigned int mask = 1;
11
       puts("The result of combining the following");
12
13
       displayBits(number1);
       displayBits(mask);
14
       puts("using the bitwise AND operator & is");
15
       displayBits(number1 & mask);
16
17
       // demonstrate bitwise inclusive OR (|)
18
       number1 = 15;
19
       unsigned int setBits = 241;
20
       puts("\nThe result of combining the following");
21
22
       displayBits(number1);
23
       displayBits(setBits);
       puts("using the bitwise inclusive OR operator | is");
24
25
       displayBits(number1 | setBits);
26
27
       // demonstrate bitwise exclusive OR (^)
28
       number1 = 139;
       unsigned int number2 = 199;
29
       puts("\nThe result of combining the following");
30
31
       displayBits(number1);
32
       displayBits(number2);
       puts("using the bitwise exclusive OR operator ^ is");
33
       displayBits(number1 ^ number2);
34
35
```

Fig. 10.5 Using the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 1 of 2.)

```
36
      // demonstrate bitwise complement (~)
37
      number1 = 21845;
      puts("\nThe one);
38
39
      displayBits(number1);
40
      puts("is");
41
      displayBits(~number1);
42
   }
43
   // display bits of an unsigned int value
44
   void displayBits(unsigned int value) {
45
      // declare displayMask and left shift 31 bits
46
      unsigned int displayMask = 1 << 31;</pre>
47
48
      printf("%10u = ", value);
49
50
51
      // loop through bits
52
      for (unsigned int c = 1; c \le 32; ++c) {
53
         putchar(value & displayMask ? : );
         value <<= 1; // shift value left by 1
54
55
56
         if (c \% 8 == 0) \{ // \text{ output a space after } 8 \text{ bits} 
57
            putchar();
58
         }
59
      }
60
61
      putchar();
62 }
The result of combining the following
     65535 = 00000000 00000000 11111111 11111111
         using the bitwise AND operator & is
         The result of combining the following
       15 = 00000000 \ 00000000 \ 00000000 \ 00001111
       241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
       The result of combining the following
       139 = 00000000 00000000 00000000 10001011
       199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
       76 = 00000000 00000000 00000000 01001100
The one
     21845 = 00000000 00000000 01010101 01010101
4294945450 = 11111111 11111111 10101010 10101010
```

Fig. 10.5 Using the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 2 of 2.)

Bitwise AND Operator (&)

Line 10 assigns the value 65535

00000000 00000000 11111111 11111111

to the integer variable number1, and line 11 assigns the value 1

00000000 00000000 00000000 00000001

to the variable mask. When you combine number1 and mask using the bitwise AND operator (&) in the expression number1 & mask (line 16), the result is

00000000 00000000 00000000 00000001

All the bits except the low-order bit in number1 are "masked off" (hidden) by "ANDing" with variable mask.

Bitwise Inclusive OR Operator (|)

The bitwise inclusive OR operator sets specific bits to 1 in an operand. Line 19 assigns 15

00000000 00000000 00000000 00001111

to the variable number1 and line 20 assigns 241

00000000 00000000 00000000 11110001

to the variable setBits. When you combine number1 and setBits with the bitwise inclusive OR operator in the expression number1 \mid setBits (line 25), the result is 255

00000000 00000000 00000000 11111111

The following table summarizes the results of combining two bits with the bitwise inclusive OR operator.

Bit I	Bit 2	Bit 1 Bit 2
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise Exclusive OR Operator (^)

The bitwise exclusive OR operator (^) sets each bit in the result to 1 if exactly one of the corresponding bits in its two operands is 1. Line 28 assigns number1 the value 139

00000000 00000000 00000000 10001011

and line 29 assigns number2 the value 199

00000000 00000000 00000000 11000111

When you combine these variables with the bitwise exclusive OR operator in the expression number1 ^ number2 (line 34), the result is

00000000 00000000 00000000 01001100

The following table summarizes the results of combining two bits with the bitwise exclusive OR operator.

Bit I	Bit 2	Bit 1 ^ Bit 2
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise Complement Operator (~)

The bitwise complement operator (~) sets all 1 bits in its operand to 0 in the result and sets all 0 bits to 1. This is otherwise referred to as "taking the one's complement of the value." Line 37 assigns number1 the value 21845

```
00000000 00000000 01010101 01010101
```

The expression ~number1 (line 41) toggles all the bits producing

11111111 11111111 10101010 10101010

10.9.4 Using the Bitwise Left- and Right-Shift Operators

Figure 10.6 demonstrates the left-shift (<<) and right-shift (>>) operators. Again, we use the function displayBits to display the unsigned int values.

```
I // fig10_06.c
 2 // Using the bitwise shift operators
 3 #include <stdio.h>
 5 void displayBits(unsigned int value); // prototype
 7
    int main(void) {
 8
       unsigned int number1 = 960; // initialize number1
 9
10
       // demonstrate bitwise left shift
П
       puts("\nThe result of left shifting");
12
       displayBits(number1);
       puts("8 bit positions using the left shift operator << is");</pre>
13
14
       displayBits(number1 << 8);</pre>
15
16
       // demonstrate bitwise right shift
17
       puts("\nThe result of right shifting");
18
       displayBits(number1);
19
       puts("8 bit positions using the right shift operator >> is");
20
       displayBits(number1 >> 8);
21
    }
22
```

Fig. 10.6 Using the bitwise shift operators. (Part 1 of 2.)

```
// display bits of an unsigned int value
23
    void displayBits(unsigned int value) {
24
25
       // declare displayMask and left shift 31 bits
26
       unsigned int displayMask = 1 << 31;</pre>
27
       printf("%10u = ", value);
28
29
       // loop through bits
30
31
       for (unsigned int c = 1; c \leftarrow 32; ++c) {
          putchar(value & displayMask ? : );
32
          value <<= 1; // shift value left by 1
33
34
          if (c % 8 == 0) { // output a space after 8 bits
35
36
             putchar();
37
          }
38
       }
39
40
       putchar();
    }
41
The result of left shifting
       960 = 00000000 00000000 00000011 11000000
8 bit positions using the left shift operator << is
    245760 = 00000000 00000011 11000000 00000000
The result of right shifting
       960 = 00000000 00000000 00000011 11000000
8 bit positions using the right shift operator >> is
         3 = 00000000 00000000 00000000 00000011
```

Fig. 10.6 Using the bitwise shift operators. (Part 2 of 2.)

Left-Shift Operator (<<)

The left-shift operator (<<) shifts the bits of its left operand to the left by the number of bits specified in its right operand. Bits vacated to the right are replaced with 0s. Bits shifted off the left are lost. Line 8 assigns the variable number1 the value 960

```
00000000 00000000 00000011 11000000
```

Left-shifting number1 eight bits with the expression number1 << 8 (line 14) results in the value 245760

00000000 00000011 11000000 00000000

Right-Shift Operator (>>)

The right-shift operator (>>) shifts the bits of its left operand to the right by the number of bits specified in its right operand. Right-shifting an unsigned int replaces the vacated bits at the left with 0s. Bits shifted off the right are lost. The result of right-shifting number1 with the expression number1 >> 8 (line 20) is 3

00000000 00000000 00000000 00000011

The result of right- or left-shifting a value is undefined if the right operand is negative $\bigotimes \mathsf{ERR}$ or if the right operand is larger than the number of bits in the left operand. The result of right-shifting a negative number is implementation-defined.



10.9.5 Bitwise Assignment Operators

Each binary bitwise operator has a corresponding assignment operator. The following table summarizes these bitwise assignment operators.

Bitwise assignment operators			
& =	Bitwise AND assignment operator.		
=	Bitwise inclusive OR assignment operator.		
^=	Bitwise exclusive OR assignment operator.		
<<=	Left-shift assignment operator.		
>>=	Right-shift assignment operator.		

The following table shows in decreasing order the precedence and grouping of the operators introduced to this point in the text.

Operator	Grouping Type
() []> ++ (postfix) (postfix)	left-to-right highest
+ - ++! & * ~ sizeof (<i>type</i>)	right-to-left unary
* / %	left-to-right multiplicative
+ -	left-to-right additive
<< >>	left-to-right shifting
< <= > >=	left-to-right relational
== !=	left-to-right equality
&	left-to-right bitwise AND
٨	left-to-right bitwise XOR
	left-to-right bitwise OR
&&	left-to-right logical AND
H	left-to-right logical OR
?:	left-to-right conditional
= += -= *= /= %= &= = ^= <<= >>=	left-to-right assignment
,	left-to-right comma

Self Check

(Fill-In) Often, the bitwise AND operator is used with an operand called a ____, which is an integer value with specific bits set to 1. This is used to hide some bits in a value while selecting other bits.

Answer: mask.

2 (*TruelFalse*) The bitwise exclusive OR operator sets each bit in the result to 1 if the corresponding bit in either (or both) operand(s) is 1.

Answer: *False.* Actually, what's described above is the bitwise inclusive OR operator. The bitwise exclusive OR operator sets each bit in the result to 1 if the corresponding bits in each operand are different.

- **3** (*Fill-In*) The bitwise complement operator sets all 0 bits in its operand to 1 in the result and sets all 1 bits to 0. This is often called ______ the bits. Answer: toggling.
- **4** (*True/False*) Because of the machine-dependent nature of bitwise manipulations, programs including them might not work correctly or might work differently across systems.

Answer: True.

10.10 Bit Fields

You can specify the number of bits in which to store an unsigned or signed integral member of a struct or union. Known as bit fields, these enable better memory utilization by storing data in the minimum number of bits required. Bit field members typically are declared as int or unsigned int.

10.10.1 Defining Bit Fields

The following struct bitCard

```
struct bitCard {
   unsigned int face : 4;
   unsigned int suit : 2;
   unsigned int color : 1;
};
```

contains three unsigned int bit fields—face, suit and color—that can represent a card in a deck of 52 cards. You declare a bit field by following an unsigned or signed integral member's name with a colon (:) and an integer constant representing the bit field's width—the number of bits in which to store the member. The width must be an integer constant between 0 and the total number of bits used to store an int on your system, inclusive. Our examples were tested on a computer with four-byte (32-bit) integers.

The preceding structure definition indicates that members face, suit and color are stored in 4 bits, 2 bits and 1 bit, respectively. The number of bits is based on the desired range of values for each member:

- face stores values from 0 (Ace) through 12 (King)—4 bits can store values in the range 0–15,
- suit stores values from 0 through 3 (0 = Hearts, 1 = Diamonds, 2 = Clubs, 3 = Spades)—2 bits can store values in the range 0–3, and
- color stores either 0 (Red) or 1 (Black)—1 bit can store either 0 or 1.

10.10.2 Using Bit Fields to Represent a Card's Face, Suit and Color

Figure 10.7 creates the array deck containing 52 struct bitCard structures in line 19. Function fillDeck (lines 30–37) inserts the 52 cards in the deck array, and function deal (lines 41–49) prints the 52 cards. Notice that bit field members of structures are accessed exactly as any other structure member.

```
I // fig10_07.c
 2 // Representing cards with bit fields in a struct
 3 #include <stdio.h>
 4 #define CARDS 52
 6 // bitCard structure definition with bit fields
 7
    struct bitCard {
       unsigned int face : 4; // 4 bits; 0-15
 8
 9
       unsigned int suit : 2; // 2 bits; 0-3
       unsigned int color : 1; // 1 bit; 0-1
10
11
    };
12
13
   typedef struct bitCard Card; // new type name for struct bitCard
14
void fillDeck(Card * const deck); // prototype
16 void deal(const Card * const deck); // prototype
17
18 int main(void) {
       Card deck[CARDS]; // create array of Cards
19
20
21
       fillDeck(deck);
22
23
       puts("Card values 0-12 correspond to Ace through King");
24
       puts("Suit values 0-3 correspond to Hearts, Diamonds, Clubs and Spades");
25
       puts("Color values 0-1 correspond to red and black\n");
26
       deal(deck);
27
   }
28
29 // initialize Cards
30 void fillDeck(Card * const deck) {
31
       // loop through deck
32
       for (size_t i = 0; i < CARDS; ++i) {
33
          deck[i].face = i \% (CARDS / 4);
          deck[i].suit = i / (CARDS / 4);
34
          deck[i].color = i / (CARDS / 2);
35
36
       }
37 }
38
39 // output cards in two-column format; cards 0-25 indexed with
   // k1 (column 1); cards 26-51 indexed with k2 (column 2)
41 void deal(const Card * const deck) {
       // loop through deck
42
43
       for (size_t k1 = 0, k2 = k1 + 26; k1 < CARDS / 2; ++k1, ++k2) {
```

Fig. 10.7 | Representing cards with bit fields in a struct. (Part 1 of 2.)

```
Card values 0-12 correspond to Ace through King
Suit values 0-3 correspond to Hearts, Diamonds, Clubs and Spades
Color values 0-1 correspond to red and black
Card:
         Suit: 0 Color: 0
                           Card: 0
                                    Suit: 2
                                           Color: 1
                                    Suit: 2 Color: 1
         Suit: 0 Color: 0
Card:
      1
                           Card: 1
Card:
         Suit: 0 Color: 0
                           Card: 2
                                    Suit: 2 Color: 1
                          Card: 3
Card: 3
         Suit: 0 Color: 0
                                    Suit: 2 Color: 1
Card: 4 Suit: 0 Color: 0
                          Card: 4 Suit: 2 Color: 1
Card: 5 Suit: 0 Color: 0
                          Card: 5 Suit: 2 Color: 1
Card: 6 Suit: 0 Color: 0
                          Card: 6
                                    Suit: 2 Color: 1
                          Card: 7
Card: 7
         Suit: 0 Color: 0
                                    Suit: 2 Color: 1
                                    Suit: 2 Color: 1
        Suit: 0 Color: 0
                          Card: 8
Card: 8
                          Card: 9
        Suit: 0 Color: 0
                                    Suit: 2 Color: 1
Card: 9
Card: 10 Suit: 0 Color: 0
                           Card: 10
                                    Suit: 2 Color: 1
Card: 11
         Suit: 0 Color: 0
                          Card: 11
                                    Suit: 2 Color: 1
                                    Suit: 2 Color: 1
Card: 12
         Suit: 0 Color: 0
                          Card: 12
                                    Suit: 3 Color: 1
         Suit: 1 Color: 0
Card: 0
                          Card: 0
Card:
         Suit: 1 Color: 0
                          Card: 1
                                    Suit: 3 Color: 1
      1
                          Card: 2
Card: 2
         Suit: 1 Color: 0
                                    Suit: 3 Color: 1
Card: 3 Suit: 1 Color: 0
                          Card: 3 Suit: 3 Color: 1
Card: 4 Suit: 1 Color: 0
                          Card: 4 Suit: 3 Color: 1
Card: 5
        Suit: 1 Color: 0
                          Card: 5
                                    Suit: 3 Color: 1
Card: 6 Suit: 1 Color: 0
                          Card: 6
                                    Suit: 3 Color: 1
                                    Suit: 3 Color: 1
      7
         Suit: 1 Color: 0
                          Card: 7
Card:
Card: 8 Suit: 1 Color: 0
                          Card: 8
                                    Suit: 3 Color: 1
Card: 9 Suit: 1 Color: 0
                           Card: 9
                                    Suit: 3 Color: 1
Card: 10 Suit: 1 Color: 0
                           Card: 10
                                    Suit: 3 Color: 1
Card: 11 Suit: 1 Color: 0
                           Card: 11
                                    Suit: 3 Color: 1
Card: 12 Suit: 1 Color: 0
                           Card: 12
                                    Suit: 3 Color: 1
```

Fig. 10.7 Representing cards with bit fields in a struct. (Part 2 of 2.)

Bit fields can reduce the amount of memory a program needs, but are machine-dependent. Although bit fields save space, using them can cause the compiler to generate slower-executing machine-language code. This occurs because it takes extra machine-language operations to access only portions of an addressable storage unit.

PERF : This is one of many examples of the kinds of space/time trade-offs that occur in computer science.

Bit fields do not have addresses, so attempting to take the address of a bit field ERR with the & operator is an error. Also, using sizeof with a bit field is an error.

10.10.3 Unnamed Bit Fields

An unnamed bit field is used as padding in a struct. For example, the definition

```
struct example {
  unsigned int a : 13;
  unsigned int : 19;
  unsigned int b : 4;
};
```

uses an unnamed 19-bit field as padding. Nothing can be stored in those 19 bits. Member b (assuming a four-byte-word computer) is stored in a separate word of memory.

An unnamed bit field with a zero width aligns the next bit field on a new storageunit boundary. For example, the struct

```
struct example {
  unsigned int a : 13;
  unsigned int : 0;
  unsigned int : 4;
};
```

uses an unnamed 0-bit field to skip the remaining bits (as many as there are) of the storage unit in which a is stored and to align b on the next storage-unit boundary.

✓ Self Check

I (*Fill-In*) The structure definition

```
struct example {
  unsigned int a : 13;
  unsigned int : 19;
  unsigned int b : 4;
};
```

uses an unnamed 19-bit field as ______mothing can be stored in those 19 bits. Answer: padding.

- **2** (*Multiple Choice*) Which of the following statements a), b) or c) about Section 10.10.1's struct bitCard is *false*?
 - a) A bit field is declared by following an unsigned or signed integral member name with a colon (:) and an integer constant representing the bit field's width.
 - b) The struct bitCard definition indicates that member face is stored in 4 bits, member suit is stored in 2 bits, and member color is stored in 1 bit.
 - c) The number of bits in a bit field is based on each structure member's desired range of values.
 - d) All of the above statements are *true*.

Answer: d.

10.11 Enumeration Constants

Section 5.11 introduced the keyword enum for defining a set of integer enumeration constants represented by identifiers. Values in an enum start with 0, unless specified otherwise, and increment by 1. For example, the enumeration

```
enum months {
    JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
};
```

creates the new type enum months in which the identifiers are set to the integers 0 through 11. To number the months 1 to 12, use:

```
enum months {
    JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
};
```

which explicitly sets JAN to 1. The remaining values increment from 1, resulting in the values 1 through 12.

The identifiers in any enumeration accessible in a given scope must be unique. Each enumeration constant's value can be set explicitly in the definition by assigning a value to the identifier. Multiple enumeration members can have the same constant value. ERR (X) Assigning a value to an enumeration constant after it's been defined is a syntax error. You should use only uppercase letters in enumeration constant names to make them stand out in a program and as a reminder that enumeration constants are not variables.

Figure 10.8 uses the enumeration variable month in a for statement to print the months of the year from the array monthName. We set monthName[0] to the empty string "" and ignore it in this example.

```
I // fig10_08.c
 2 // Using an enumeration
 3 #include <stdio.h>
 5
    // enumeration constants represent months of the year
 6
    enum months {
        JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
 7
 8
    };
 9
10 int main(void) {
        // initialize array of pointers
11
        const char *monthName[] = { "", "January", "February", "March",
    "April", "May", "June", "July", "August", "September", "October",
12
13
           "November", "December" };
14
15
16
        // loop through months
        for (enum months month = JAN; month <= DEC; ++month) {</pre>
17
18
           printf("%2d%11s\n", month, monthName[month]);
19
        }
20 }
 1
       January
 2
      February
 3
         March
 4
         April
 5
           May
 6
          June
```

Fig. 10.8 | Using an enumeration. (Part 1 of 2.)

```
7 July
8 August
9 September
10 October
11 November
12 December
```

Fig. 10.8 | Using an enumeration. (Part 2 of 2.)

Self Check

I (*Code*) The following enumeration creates a new type, enum days, in which the identifiers are set to the integers 0 to 6:

```
enum days {
   MON, TUE, WED, THU, FRI, SAT, SUN
};
```

Rewrite this enumeration to number the days 1 to 7.

Answer:

```
enum days {
   MON = 1, TUE, WED, THU, FRI, SAT, SUN
:
```

2 (*True/False*) Multiple enumeration constants in the same scope can have the same identifier.

Answer: False. Actually, the identifiers in any enumeration accessible in the same scope must be unique. Multiple members of an enumeration can have the same constant value.

10.12 Anonymous Structures and Unions

Anonymous structs and unions can be nested in named structs and unions. The members in a nested anonymous struct or union are members of the enclosing struct or union. They can be accessed directly through an object of the enclosing type. For example, consider the following struct declaration:

```
struct myStruct {
   int member1;
   int member2;

   struct { # anonymous struct
      int nestedMember1;
      int nestedMember2;
   }; // end nested struct
}; // end outer struct
```

For a struct myStruct variable named object, you can access the members as

```
object.member1;
object.member2;
object.nestedMember1;
object.nestedMember2;
```



Self Check

(True/False) The members in a nested anonymous struct or union are members of the enclosing struct or union. They can be accessed directly through an object of the enclosing type.

Answer: True.



SEC 10.13 Secure C Programming

Various CERT guidelines and rules apply to this chapter's topics. For more information on each, visit https://wiki.sei.cmu.edu/.

CERT Guidelines for structs

As we discussed in Section 10.2.4, the boundary alignment requirements for struct members may result in extra bytes containing undefined data for each struct variable you create. Each of the following guidelines is related to this issue:

- EXP03-C: Because of boundary alignment requirements, a struct variable's size is *not* necessarily the sum of its members' sizes. Always use sizeof to determine a struct variable's number of bytes. We'll use this technique to manipulate fixed-length records that are written to and read from files (Chapter 11) and to create custom data structures (Chapter 12).
- EXP04-C: Section 10.2.4 discussed that struct variables cannot be compared for equality or inequality because they might contain bytes of undefined data. Therefore, you must compare their individual members.
- DCL39-C: In a struct variable, the undefined extra bytes could contain secure data—left over from prior use of those memory locations—that should not be accessible. This CERT guideline discusses compiler-specific mechanisms for packing the data to eliminate these extra bytes.

CERT Guideline for typedef

DCL05-C: Complex type declarations, such as those for function pointers, can be difficult to read. You should use typedef to create self-documenting type names that make your programs more readable.

CERT Guidelines for Bit Manipulation

- INT02-C: As a result of the integer promotion rules (discussed in Section 5.6), performing bitwise operations on integer types smaller than int can lead to unexpected results. Explicit casts are required to ensure correct results.
- INT13-C: Some bitwise operations on signed integer types are implementation-defined—this means that the operations may have different results across C compilers. For this reason, unsigned integer types should be used with the bitwise operators.

EXP46-C: The logical operators && and || are frequently confused with the bitwise operators & and |, respectively. Using & and | in a conditional expression's condition (?:) can lead to unexpected behavior because the & and | operators do not use short-circuit evaluation.

CERT Guideline for enum

INT09-C: Allowing multiple enumeration constants to have the same value can result in difficult-to-find logic errors. In most cases, an enum's enumeration constants should each have unique values to help prevent such logic errors.

Self Check

- (Fill-In) struct variables cannot be compared for equality or inequality, because they might contain bytes of undefined data. Instead, you must _____. **Answer:** compare their individual members.
- (True/False) Allowing multiple enumeration constants to have the same value can result in difficult-to-find logic errors. In most cases, an enum's enumeration constants should each have unique values to help prevent such logic errors. Answer: True.

Summary

Section 10.1 Introduction

- Structures (p. 482) are collections of related variables under one name. They may contain variables of many different data types.
- Structures are commonly used to define records to be stored in files.
- Pointers and structures can be used to form more complex data structures, such as linked lists, queues, stacks and trees.

Section 10.2 Structure Definitions

- Keyword **struct** introduces a structure definition (p. 483).
- The structure tag (p. 483) following keyword struct names the structure definition. It's used with the keyword struct to declare variables of the struct type.
- Variables declared within the braces of a struct definition are the struct's members.
- Members of the same struct type must have unique names.
- Each struct definition must end with a semicolon.
- struct members can have primitive or aggregate data types.
- A struct cannot contain an instance of itself but may include a pointer to its type.
- A struct containing a member that's a pointer to the same struct type is referred to as a self-referential structure. Self-referential structures (p. 483) are used to build linked data structures.
- struct definitions create new data types that are used to define variables.
- Variables of a given struct type can be declared by placing a comma-separated list of variable names between the struct definition's closing brace and its ending semicolon.

- If a struct definition does not contain a structure tag name, variables of the struct type may be declared only in the struct definition.
- The only valid operations that may be performed on structs are assigning struct variables to variables of the same type, taking the address (&) of a struct variable, accessing the members of a struct variable and using the sizeof operator to determine the size of a struct variable.

Section 10.3 Initializing Structures

- structs can be initialized using initializer lists.
- If there are fewer initializers in the list than members in the struct, the remaining members are automatically initialized to 0 (or NULL if the member is a pointer).
- Members of struct variables defined outside a function definition are initialized to 0 or NULL if they're not explicitly initialized in the external definition.

Section 10.4 Accessing Structure Members with . and ->

- The structure member operator (.) and the structure pointer operator (->) are used to access structure members (p. 486).
- The structure member operator accesses a structure member via a struct variable name.
- The structure pointer operator accesses a struct member via a pointer to a struct object (p. 486).

Section 10.5 Using Structures with Functions

- struct members, entire struct objects or pointers to struct objects may be passed to functions.
- Entire struct objects are passed by value by default.
- To pass a struct object by reference, pass its address. Arrays of struct objects are automatically passed by reference.
- To pass an array by value, create a struct with the array as a member. structs are passed by value, so the array is passed by value.

Section 10.6 typedef

- The keyword **typedef** (p. 488) creates synonyms for previously defined types.
- Names for structure types are often defined with typedef to create shorter type names.

Section 10.8 Unions

- A union (p. 492) is declared with the keyword union. Its members share the same storage space.
- A union's members can be of any data type. Operator sizeof will always return a value at least as large as the size in bytes of the union's largest member
- Only one union member can be referenced at a time. It's your responsibility to access the currently stored member.
- The valid operations on a union are assigning a union to another of the same type, taking the address (&) of a union variable, and accessing union members using the structure member operator and the structure pointer operator.
- A union may be initialized in a declaration with a value of the first union member's type.

Section 10.9 Bitwise Operators

- Computers represent all data internally as sequences of bits with the values 0 or 1.
- On most systems, a sequence of **8 bits form a byte**—the standard storage unit for a variable of type char. Other data types are stored in larger numbers of bytes.
- The **bitwise operators** manipulate the bits of integral operands (char, short, int and long; both signed and unsigned). Unsigned integers are normally used.
- The bitwise operators (p. 495) are bitwise AND (&), bitwise inclusive OR (|), bitwise exclusive OR (^), left shift (<<), right shift (>>) and complement (~).
- The bitwise AND, bitwise inclusive OR and bitwise exclusive OR operators compare their two operands bit by bit. The bitwise AND operator (p. 495) sets each bit in the result to 1 if the corresponding bit in both operands is 1. The bitwise inclusive OR operator (p. 495) sets each bit in the result to 1 if the corresponding bit in either (or both) operand(s) is 1. The bitwise exclusive OR operator (p. 495) sets each bit in the result to 1 if the corresponding bits in both operands are different.
- The **left-shift operator** (p. 495) shifts the bits of its left operand to the left by the number of bits specified in its right operand. Bits vacated to the right are replaced with 0s; bits shifted off the left are lost.
- The **right-shift operator** (p. 495) shifts the bits in its left operand to the right by the number of bits specified in its right operand. Performing a right shift on an unsigned int causes the vacated bits at the left to be replaced by 0s; bits shifted off the right are lost.
- The bitwise complement operator (p. 495) sets all 0 bits in its operand to 1 and all 1 bits to 0 in the result.
- Often, bitwise AND is used with an operand called a mask (p. 497)—an integer value with specific bits set to 1. Masks are used to hide some bits in a value while selecting other bits.
- **CHAR_BIT** (p. 497; defined in <1imits.h>) represents the number of bits in a byte (normally 8). It can be used to make a bit-manipulation program more generic and portable.
- Each binary bitwise operator has a corresponding bitwise assignment operator (p. 503).

Section 10.10 Bit Fields

- A bit field (p. 504) specifies the number of bits in which an unsigned or signed integral member of a structure or union is stored.
- A bit field is declared by following an unsigned int or int member name with a colon (:) and an integer constant representing the width of the field (p. 504). The constant must be an integer between 0 and the total number of bits used to store an int on your system, inclusive.
- Bit-field members of structures are accessed exactly as any other structure member.
- It's possible to specify an unnamed bit field (p. 507) to be used as padding in a structure (p. 507).
- An unnamed bit field with a zero width (p. 507) aligns the next bit field on a new storageunit boundary.

Section 10.11 Enumeration Constants

- An **enum** defines a set of integer constants represented by identifiers (p. 507). Values in an enum start with 0, unless specified otherwise, and are incremented by 1.
- The identifiers in an enum must be unique.
- The value of an enum constant can be set explicitly via assignment in the enum definition.

Self-Review Exercises

10.1	Fill-In the blanks in each of the following:
	a) A(n) is a collection of related variables under one name.
	b) A(n) is a collection of variables under one name in which the vari-
	ables share the same memory.
	c) In an expression using the operator, bits are set to 1 if the corre-
	sponding bits in each operand are 1. Otherwise, the bits are set to zero.
	d) The variables declared in a structure definition are called its
	e) In an expression using the operator, bits are set to 1 if at least one of
	the corresponding bits in either operand is 1. Otherwise, the bits are set to 0.
	f) Keyword introduces a structure declaration.
	g) Keyword creates a synonym for a previously defined data type.
	h) In an expression using the operator, bits are set to 1 if exactly one of
	the corresponding bits in either operand is 1. Otherwise, the bits are set to 0.
	i) The bitwise AND operator (&) is often used to bits—that is, to se-
	lect certain bits while zeroing others.
	j) Keyword is used to introduce a union definition.
	k) The name of the structure is referred to as the structure
	l) You access a structure member with the or operators.
	m)The and operators shift the bits of a value left or right.
	n) A(n) is a set of integers represented by identifiers.
10.2	State whether each of the following is true or false. If false, explain why.
	a) structs may contain variables of only one data type.
	b) Two unions can be compared (using ==) to determine whether they're equal.
	c) The tag name of a struct is optional.
	d) Members of different structs must have unique names.
	e) Keyword typedef is used to define new data types.
	f) structs are always passed to functions by reference.
	g) structs may not be compared by using operators == and !=.
10.3	Write code to accomplish each of the following:
	a) Define a struct called part containing unsigned int variable partNumber
	and char array partName with values that may be as long as 25 characters (in-
	cluding the terminating null character).
	b) Define Part to be a synonym for the type struct part.
	c) Use Part to declare variable a to be of type struct part, array b[10] to be of
	type struct part and variable ptr to be of type pointer to struct part.
	d) Read a part number and a part name from the keyboard into the individual
	members of variable a.
	e) Assign the member values of variable a to element 3 of array b.
	f) Assign the address of array b to the pointer variable ptr.
	g) Print the member values of element 3 of array b using the variable ptr and

the structure pointer operator to refer to the members.

- **10.4** Find the error in each of the following:
 - a) Assume that struct card contains two const char * pointers named face and suit. Also, the variable c is a struct card, and the variable cPtr is a pointer to struct card. Variable cPtr has been assigned the address of c.

```
printf("%s\n", *cPtr->face);
```

b) Assume that struct card contains two const char * pointers named face and suit. Also, the array hearts[13] is an array of type struct card. The following statement should print the member face of array element 10.

```
printf("%s\n", hearts.face);
```

```
c) union values {
    char w;
    float x;
    double y;
    };
    union values v = {1.27};
d) struct person {
    char lastName[15];
    char firstName[15];
    unsigned int age;
}
```

e) Assume struct person has been defined as in part (d) but with the appropriate correction.

```
person d;
```

f) Assume variable p has type struct person and the variable c is a struct card.

```
p = c;
```

Answers to Self-Review Exercises

10.1 a) structure. b) union. c) bitwise AND (&). d) members. e) bitwise inclusive OR (|). f) struct. g) typedef. h) bitwise exclusive OR (^). i) mask. j) union. k) tag name. l) structure member, structure pointer. m) left-shift (<<), right-shift (>>). n) enumeration.

10.2 See the answers below:

- a) False. A structure can contain variables of many data types.
- b) *False*. Unions cannot be compared because there might be bytes of undefined data with different values in union variables that are otherwise identical.
- c) True.
- d) *False*. The members of separate structures can have the same names, but the members of a given structure must have unique names.

- e) False. Keyword typedef is used to define new names (synonyms) for previously defined data types.
- f) False. Structures are always passed to functions by value.
- g) *True*, because of alignment problems.
- **10.3** See the answers below:

```
a) struct part {
     unsigned int partNumber;
     char partName[25];
  };
b) typedef struct part Part;
c) Part a, b[10], *ptr;
d) scanf("%d%24s", &a.partNumber, a.partName);
e) b[3] = a;
f) ptr = b;
g) printf("%d %s\n", (ptr + 3)->partNumber, (ptr + 3)->partName);
```

- **10.4** See the answers below:
 - a) The parentheses that should enclose *cPtr have been omitted, causing the order of evaluation of the expression to be incorrect. The expression should be

```
cPtr->face
or
   (*cPtr).face
```

- b) The array index is missing. The expression should be hearts[10].face.
- c) A union can be initialized only with a value that has the same type as the union's first member.
- d) A semicolon is required to end a structure definition.
- e) Keyword struct was omitted from the variable declaration. The declaration should be

```
struct person d;
```

f) Variables of different structure types cannot be assigned to one another.

Exercises

- **10.5** Provide the definition for each of the following structures and unions:
 - a) struct inventory containing character array partName[30], integer part-Number, floating-point price, integer stock and integer reorder.
 - b) union data containing char c, short s, long b, float f and double d.
 - c) A struct called address that contains character arrays streetAddress[25], city[20], state[3] and zipCode[6].
 - d) struct student that contains arrays firstName[15] and lastName[15] and variable homeAddress of type struct address from part (c).
 - e) struct test containing sixteen-bit fields of one bit each. The names of the bit fields are the letters a to p.

10.6 Given the following struct and variable definitions:

```
struct customer {
   char lastName[15];
   char firstName[15];
   unsigned int customerNumber;

   struct {
      char phoneNumber[11];
      char address[50];
      char city[15];
      char state[3];
      char zipCode[6];
   } personal;
} customerRecord, *customerPtr;
```

write an expression that accesses the struct members in each of the following parts:

- a) Member lastName of struct customerRecord.
- b) Member TastName of the struct pointed to by customerPtr.
- c) Member firstName of struct customerRecord.
- d) Member firstName of the struct pointed to by customerPtr.
- e) Member customerNumber of struct customerRecord.
- f) Member customerNumber of the struct pointed to by customerPtr.
- g) Member phoneNumber of member personal of struct customerRecord.
- h) Member phoneNumber of member personal of the struct pointed to by customerPtr.
- $i) \ \ Member \ address \ of \ member \ personal \ of \ struct \ customer Record.$
- Member address of member personal of the struct pointed to by customerPtr.
- Member city of member personal of the struct pointed to by customerPtr.
- $m) \\ Member \ state \ of \ member \ personal \ of \ struct \ customer \\ Record.$
- n) Member state of member personal of the struct pointed to by customerPtr.
- o) Member zipCode of member personal of struct customerRecord.
- p) Member zipCode of member personal of the struct pointed to by customerPtr.
- **10.7** (Card Shuffling and Dealing Modification) Modify Figure 10.7 to shuffle the cards using a high-performance shuffle (as shown in Fig. 10.2). Print the resulting deck in a two-column format that uses the face and suit names. Precede each card with its color.
- **10.8** (*Using Unions*) Create union integer with members char c, short s, int i and long b. Write a program that inputs values of type char, short, int and long and stores the values in union variables of type union integer. Each union variable should be printed as a char, a short, an int and a long. Do the values always print correctly?

- **10.9** (Using Unions) Create union floatingPoint with members float f, double d and long double x. Write a program that inputs values of type float, double and long double and stores the values in union variables of type union floatingPoint. Each union variable should be printed as a float, a double and a long double. Do the values always print correctly?
- **10.10** (*Right-Shifting Integers*) Write a program that right-shifts an integer variable 4 bits. The program should print the integer in bits before and after the shift operation. Does your system place 0s or 1s in the vacated bits?
- **10.11** (*Left-Shifting Integers*) Left-shifting an unsigned int by 1 bit is equivalent to multiplying the value by 2. Write function power2 that takes two integer arguments number and pow and calculates

```
number * 2pow
```

Use the shift operator to calculate the result. Print the values as integers and as bits.

- **10.12** (Packing Characters into an Integer) The left-shift operator can be used to pack four character values into a four-byte unsigned int variable. Write a program that inputs four characters from the keyboard and passes them to function packCharacters. To pack four characters into an unsigned int variable, assign the first character to the unsigned int variable, shift the unsigned int variable left by 8 bit positions and combine the unsigned variable with the second character using the bitwise inclusive OR operator. Repeat this process for the third and fourth characters. Print the characters in their bit format before and after they're packed into the unsigned int to prove that the characters are, in fact, packed correctly in the unsigned int variable.
- **10.13** (Unpacking Characters from an Integer) Using the right-shift operator, the bitwise AND operator and a mask, write function unpackCharacters that takes the unsigned int from Exercise 10.12 and unpacks it into four characters. To unpack characters from a four-byte unsigned int, combine the unsigned int with the mask 4278190080 (11111111 00000000 00000000 00000000) and right-shift the result 8 bits. Assign the resulting value to a char variable. Then combine the unsigned int with the mask 16711680 (00000000 11111111 00000000 0000000). Assign the result to another char variable. Continue this process with the masks 65280 and 255. Print the unsigned int in bits before it's unpacked, then print the characters in bits to confirm that they were unpacked correctly.
- **10.14** (Reversing the Order of an Integer's Bits) Write a program that reverses the order of the bits in an unsigned int value. The program should input the value from the user and call function reverseBits to print the bits in reverse order. Print the value in bits both before and after the bits are reversed to confirm that the bits are reversed properly.
- 10.15 (Portable displayBits Function) Modify function displayBits of Fig. 10.4 so it's portable between systems using two-byte integers and systems using four-byte

integers. [*Hint:* Use the sizeof operator to determine the size of an integer on a particular machine.]

10.16 (What's the Value of X?) The following program uses function multiple to determine if the integer entered from the keyboard is a multiple of some integer X. Examine the function multiple, then determine X's value.

```
// ex10_16.c
   // This program determines whether a value is a multiple of X.
 2
   #include <stdio.h>
 5
    int multiple(int num); // prototype
 7
    int main(void) {
 8
       int y; // y will hold an integer entered by the user
 9
10
       puts("Enter an integer between 1 and 32000: ");
       scanf("%d", &y);
11
12
13
       // if y is a multiple of X
14
       if (multiple(y)) {
15
          printf("%d is a multiple of X\n", y);
16
       }
17
       else {
          printf("%d is not a multiple of X\n", y);
18
19
20
    }
21
    // determine whether num is a multiple of X
23
    int multiple(int num) {
24
       int mask = 1; // initialize mask
25
       int mult = 1; // initialize mult
26
27
       for (int i = 1; i \le 10; ++i, mask <<= 1) {
28
          if ((num & mask) != 0) {
29
             mult = 0;
30
             break;
31
          }
       }
32
33
34
       return mult;
35
    }
```

10.17What does the following program do?

```
// ex10_17.c
#include <stdio.h>

int mystery(unsigned int bits); // prototype

int main(void) {
   unsigned int x; // x will hold an integer entered by the user
```

```
9
       puts("Enter an integer: ");
       scanf("%u", &x);
10
11
       printf("The result is %d\n", mystery(x));
12
13
    }
14
    // What does this function do?
15
16
    int mystery(unsigned int bits) {
17
       unsigned int mask = 1 << 31; // initialize mask</pre>
       unsigned int total = 0; // initialize total
18
19
       for (unsigned int i = 1; i \le 32; ++i, bits <<= 1) {
20
21
          if ((bits & mask) == mask) {
             ++total;
22
          }
23
24
       }
25
26
       return !(total % 2) ? 1 : 0;
27
    }
```

10.18 (*Fisher-Yates Shuffling Algorithm*) Research the Fisher-Yates shuffling algorithm online, then use it to reimplement the shuffle function in Fig. 10.2.

Special Section: Raylib Game-Programming Case Studies

You're about to begin an exciting and challenging journey into the worlds of graphics, animation, multimedia and game development with the free, open-source, cross-platform raylib game programming library.^{2,3} The library supports Windows, macOS, Linux and several other platforms, including Android, Raspberry Pi and the web. Raylib is a C library, but it can be used with C++, C#, Java, JavaScript, Python and many other programming languages.⁴

In this Special Section's first three case studies, you'll study two games and a simulation that we created to help you learn raylib fundamentals:

- In Exercise 10.19, you'll study our completely coded **SpotOn** game, which tests your reflexes by requiring you to click fast-moving spots before they disappear. With each new game level, the spots move even faster, making the game more challenging.
- In Exercise 10.20, you'll study our completely coded **Cannon** game, which challenges you to destroy nine moving targets before a time limit expires. A moving blocker makes the game more challenging.
- In Exercise 10.21, you'll use a dynamic visualization to make the law of large numbers "come alive." You'll study our completely coded die-rolling simulation that displays an animated bar chart. As the simulation rolls the die, it

^{2.} Raylib is Copyright ©2013-2020 Ramon Santamaria (@raysan5).

^{3. &}quot;raylib." Accessed November 14, 2020. https://www.raylib.com.

^{4. &}quot;raylib bindings." Accessed December 14, 2020. https://github.com/raysan5/raylib/blob/master/BINDINGS.md.

updates the frequencies in an array. Then, it displays each die face's frequency, its percentage of the total rolls and a bar representing the frequency's magnitude. For a six-sided die, the values 1 through 6 should each occur with "equal likelihood"—the probability of each is $1/6^{th}$ or 16.67%. If we rolled a die 6000 times, we'd expect about 1000 of each face. Like coin tossing, die rolling is random, so some faces could occur fewer or more than 1000 times. As the number of die rolls increases, you'll watch the frequencies approach 16.67% and the bars in the bar chart become nearly identical in length, confirming the law of large numbers.

These games and simulations use many raylib capabilities—shapes, text, colors, sounds, animation, collision detection and handling user-input events (such as mouse clicks and keystrokes). Each exercise suggests improvements you can make to our code.

Studying Our Complete Code Solutions

A key aspect of becoming a professional programmer is reading and understanding lots of other people's code. You'll frequently visit sites like GitHub.com looking for open-source code that you can incorporate into your own projects. For these first three raylib case studies, we provide fully coded solutions in the raylib subfolder with the chapter's example code that you download from

https://deitel.com/c-how-to-program-9-e

Each source-code file includes extensive comments that:

- overview the code's top-level functions,
- list the raylib functions we use, and
- provide details you need to understand how each program works.

You should compile, run and play with each and carefully study our code. This will be challenging but rewarding. You'll work with the cool, open-source raylib package, taking a nice leap into computer graphics and game programming. You'll then have a good foundation for attempting our suggested code modifications and other game-programming exercises.

Raylib Sample Code

The raylib development team provides many C programming demos at

https://www.raylib.com/examples.html

and sample games at

```
https://www.raylib.com/games.html
```

with complete source code. Consider studying the complete source code provided with raylib for each of these examples and games to learn other raylib features and techniques.

Implementing Your Own Raylib Games and Simulations

Using what you learn from our code in Exercises 10.19–10.21, you'll enhance our raylib games and simulation and begin creating your own:

- In Exercise 10.22, you'll reimplement your solution to The Tortoise and the Hare Race from Exercise 5.54. You'll incorporate the sounds of a traditional horse race, an image of a tortoise and an image of a hare, and you'll play the William Tell Overture in the background during the race.
- In Exercise 10.23, you'll reimplement Section 10.7's high-performance card shuffling and dealing simulation using raylib and attractive public-domain card images to display a deck of cards.
- In Exercises 10.26 and 10.27, you'll attempt enhancements to our SpotOn and Cannon games.
- In Exercises 10.28–10.30, you'll create visualizations for coin tossing, rolling two six-sided dice (producing the sums 2–12) and showing the win/loss results for the casino dice game Craps, based on the lengths of the games.

Subsequent exercises propose various other games. Get creative—design and build your own games too!

Self-Contained Raylib Windows Environment

Raylib has a self-contained Windows environment with everything you need to create your own games using raylib. The bundle contains:

- the raylib game-programming library,
- the raylib examples and sample games,
- the gcc compiler in MinGW⁵ (Minimalist GNU for Windows), and
- the Notepad++ text editor, which is preconfigured to enable you to compile and run the raylib example code, raylib sample games and your own games.

You can download the MinGW version of this self-contained environment for free from

https://raysan5.itch.io/raylib

Compiling and running the raylib examples and sample games in this environment is as simple as opening the C file in Notepad++ and pressing the F6 key. This displays a window in which you'll see the compilation and execution commands that will run when you click **OK**. For applications that do not have command-line arguments, simply click **OK** to compile and run your code. For applications with command-line arguments, such as our die-rolling simulation, modify the **Execute program** command to place the command-line arguments at the end of the line, then click **OK**.

^{5. &}quot;MinGW (Minimalist GNU for Windows)." Accessed December 16, 2020. http://www.mingw.org/.

Installing Raylib on Windows, macOS and Linux

The following URLs contain raylib download and install instructions for Windows, macOS and Linux. Windows users who choose the self-contained environment option do not need to perform these additional install instructions:

- Windows (for those who wish to use raylib with other Windows compilers): https://github.com/raysan5/raylib/wiki/Working-on-Windows
- macOS: https://github.com/raysan5/raylib/wiki/Working-on-macOS
- Linux: https://github.com/raysan5/raylib/wiki/Working-on-GNU-Linux

Raylib Cheatsheet

Though raylib is relatively easy to use, its functions are not extensively documented on raylib.com. For a complete list of raylib's functions, see the raylib cheat sheet:

https://www.raylib.com/cheatsheet/cheatsheet.html

Each function is listed with its prototype followed by a comment that briefly explains its purpose. The cheat sheet also contains the names of raylib's custom types and color constants. You'll notice that raylib's functions are named with a capital first letter. This differs from the C convention of starting function names with a lowercase first letter.

raylib.h Header on GitHub

When working with open-source software, occasionally, you may need to look at the source code to get your questions answered. For instance, raylib defines many of its own types—typically as structs or enums. Most of these are not listed in the cheat-sheet. However, the full raylib source code is available in its GitHub repository:

https://github.com/raysan5/raylib/

The header raylib.h contains the raylib type definitions:

https://github.com/raysan5/raylib/blob/master/src/raylib.h

Some of the raylib types you'll use include:

- **Vector2**: Contains x and y members to represent an *x-y* coordinate pair.
- **Rectangle:** Contains x, y, width and height members to represent the upper-left corner, width and height of a rectangle.
- Color: Colors in raylib are defined using RGBA colors. Each color has red (r), green (g), blue (b) and alpha (a; transparency) components with values in the range 0–255. See the raylib cheat sheet for a list of raylib's predefined color constants. You may also specify custom colors by creating Color objects and setting their r, g, b and a members.
- **Sound:** Contains members for storing sounds loaded into memory with ray-lib's LoadSound function.
- **Texture2D**: Contains members representing a texture loaded into graphics processing unit (GPU) memory.

For these raylib case studies, you do not need to know the Sound and Texture2D type details. If you're curious, you can view their definitions in raylib.h.

Raylib Uses Frame-By-Frame Animations

In a raylib game, a game loop drives a frame-by-frame animation. Each loop iteration performs two steps:

- 1. Update the game elements for the next animation frame: In this step, you implement the game logic that determines the game elements' new states. This is where the game-play logic is implemented. Tasks performed here include updating element positions, checking for user-input events (such as mouse clicks), detecting collisions between game elements, updating the score, checking wheter the game is over, etc. Element positions are specified as *x-y* coordinate pairs within the screen's width and height—0,0 is the upper-left corner.
- 2. Draw the next animation frame's game elements: In this step, you use raylib's drawing functions to draw the game's elements at their current positions. Raylib stores the pixels of the new animation frame in memory—known as an off-screen buffer. When the drawing step completes, raylib displays the off-screen buffer's contents, replacing the previous animation frame on the screen.

Raylib Game Structure

A typical raylib game has the following structure in its main function, which we explain below the code listing:

```
int main(void) {
       // initialization
2
3
       InitWindow(screenWidth, screenHeight, "Window Title");
4
       InitGame();
 5
       SetTargetFPS(60);
6
7
       // game loop
8
       while (!WindowShouldClose()) {
9
          UpdateGame(); // update game elements
          DrawGame(); // draw next animation frame
10
       }
11
12
13
       // cleanup
       UnloadGame(); // release game resources
14
       CloseWindow(); // close game window
15
16
   }
```

- The raylib function **InitWindow** (line 3) specifies the game window's width in pixels, height in pixels and title.
- A typical raylib sample game contains a user-defined InitGame function (line 4). This is where you load sounds, textures and images, initialize the game elements and initialize the variables that maintain the game's state. When a game terminates and the user chooses to play again, you typically call InitGame to reset the game state before starting a new game.
- The raylib function SetTargetFPS (line 5) specifies the number of animation frames raylib tries to draw each second—higher frame rates produce smoother

animations. Today's console games typically try to display 60 frames per second, though some games use more and some fewer. A minimum of 30FPS is recommended for smooth animation.

- The main game loop (lines 8–11) drives the game updates and animation. This loop runs until raylib's WindowShouldClose function returns true—when the user closes the window or presses the *Esc* key. This loop updates the game elements, then draws them. Most raylib sample games place the updating code in a function named UpdateGame and the drawing code in a function named DrawGame. This makes the code easier to maintain.
- When the game loop terminates, UnloadGame (line 14) unloads any game resources you loaded in InitGame, such as sounds, textures and images.
- Raylib function CloseWindow (line 15) releases the game window's resources and closes the game window. Then, the application terminates.

In this code, InitGame, UpdateGame, DrawGame and UnloadGame are user-defined functions that define game logic. The raylib code examples and sample games tend to use these names, which follow the capital first letter naming convention used for raylib's functions. We use the same names or similar names in our games and similar names in our simulations that are not games (e.g., InitSimulation rather than Init-Game). We define any other supporting functions with our usual function-naming conventions used throughout this book.

Global Variables and Constants

■

PERF

For performance, raylib games define the game elements and game state variables as static global variables. Such variables are known only from their definitions until the end of the file in which they're defined. Using static global variables enables the game's functions to access the game's elements and state directly without passing them to the functions as arguments. This eliminates the overhead of the function call/return mechanisms. As you'll see, even relatively simple games tend to have many game elements and game-state variables. Defining functions with large numbers of parameters tends to make the code harder to maintain, modify and debug.

How to Approach These Case Study Exercises

For each of the first three raylib exercises, we describe what the game does and show screen captures of the game in action. For each game, you should:

- 1. Read the exercise description to get a sense of the game or simulation.
- 2. Compile then run the game or simulation several times. For the games, play them to get a feel for how they work.
- 3. Immerse yourself in the fully coded and commented programs we provide.
- **4.** Tweak the code and rerun it to see the effects of your modifications.

Generally, our code starts with comments that overview the game's functions that we wrote, summarize the raylib functions we use and more.

Interacting with the Raylib Community

Here are some key sites⁶ where you can interact with other raylib users and watch raylib videos:

- Discord: https://discord.gg/VkzNHUE
- Twitter: http://www.twitter.com/raysan5
- Twitch: http://www.twitch.tv/raysan5
- Reddit: https://www.reddit.com/r/raylib
- Patreon: https://www.patreon.com/raylib
- YouTube: https://www.youtube.com/c/raylib

Raylib rFXGen Sound-Effect Generator

Raylib has several online tools that help you create items for your games, including icons, textures, graphical user interface elements and layouts and sound effects:

https://raylibtech.itch.io/

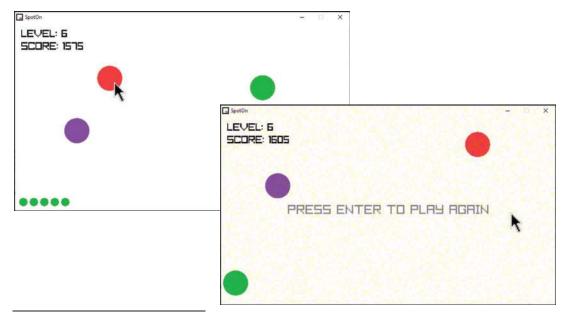
We used raylib's rFXGen online sound-effect generator:

https://raylibtech.itch.io/rfxgen

to create sound effects for our games. You can use the sound effects we provided or create your own.

Game-Programming Case Study Exercise: SpotOn Game

10.19 (Game Programming Case Study: SpotOn Game) In this game-programming case study exercise, you'll study our SpotOn game, which tests your reflexes by requiring you to click fast-moving spots before they disappear:



^{6. &}quot;README.md." Accessed December 16, 2020. https://github.com/raysan5/raylib/README.md.

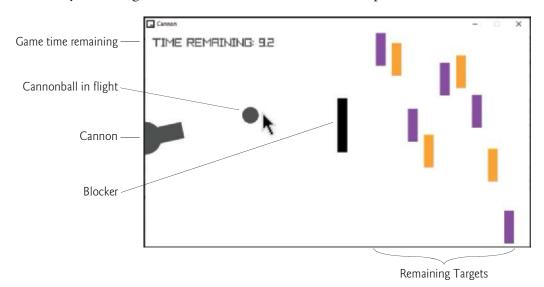
The game begins on level one by displaying three colored spots at random locations. These move at random speeds in random directions. You reach a new level for every 10 spots you click—this increases the spot speed by 5%, making the game increasingly challenging. When you click a spot, the app makes a popping sound, and the spot disappears. You receive points (10 times the current level) for each clicked spot. Accuracy is essential—any click that misses a spot plays a raspberry sound and decreases the score by 15 times the current level. Your current level and score are displayed in the game's top-left corner.

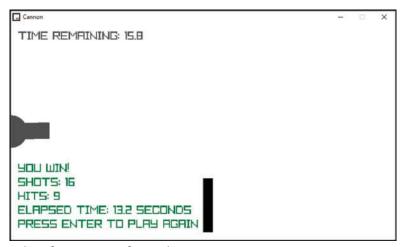
You begin the game with three lives—displayed as small circles in the game's bottom-left corner. If a spot disappears before you click it, you hear a whoosh sound and lose a life. You gain a life for each new level reached, up to a maximum of seven lives. When you lose all your lives, the game ends. You may pause the game at any time by pressing the *P* key, and resume the game by pressing *P* again.

Compile and run the game and play it several times. Next, study this game's code (including extensive comments). Consider tweaking the code to see how your changes affect gameplay. For example, you can change the spotSpeed constant's value to make the spots move faster or slower. Finally, improve the game by implementing the enhancements we suggest in Exercise 10.26.

Game-Programming Case Study: Cannon Game

10.20 (Game Programming Case Study: Cannon Game) In the Cannon game, you must destroy nine targets before a ten-second time limit expires:





The game has four types of visual components:

- a cannon that you control,
- a cannonball,
- nine targets that move up and down at various speeds, and
- a blocker that moves up and down, defending the targets.

The targets and the blocker move vertically at different but fixed speeds, reversing direction when they hit the screen's top or bottom.

To fire the cannon, you click the mouse. The cannon rotates toward the click point, fires a fast-moving cannonball in a straight line in that direction and plays a **boom sound**. Only one cannonball can be on the screen at a time.

Each time you destroy a target, a **target-destruction sound** plays, the target disappears, and the time remaining increases by a three-second time bonus. The blocker cannot be destroyed. When the cannonball hits the blocker, a **blocker-hit sound** plays, the cannonball bounces back, and the time remaining decreases by a two-second time penalty.

You win by destroying all nine target sections before the time expires. If the timer reaches zero, you lose. At the end of the game, the app displays whether you won or lost and shows the number of shots fired and the elapsed time. You may pause the game at any time by pressing the *P* key, and resume the game by pressing *P* again.

Compile and run the game and play it several times. Next, study this game's code and extensive comments. This application requires some trigonometry to:

- determine the cannon barrel's endpoint, based on its angle, and
- determine the cannonball's x and y increments used to move the cannonball
 in each animation frame—these also are based on the cannon's barrel angle.

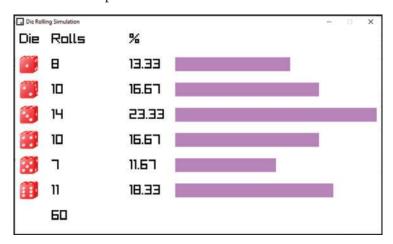
We provide the trigonometry calculations for you.

Consider tweaking the code to see how your changes affect gameplay. For example, you could change how fast the cannonball moves. Finally, improve the game by implementing the enhancements we suggest in Exercise 10.27.

Visualization with raylib—Law of Large Numbers Animation

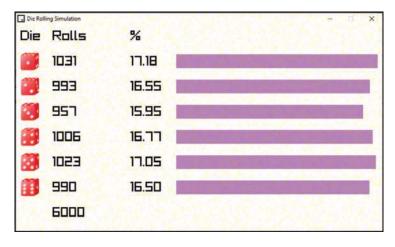
10.21 (*Law of Large Numbers Animation*) In Sections 5.10 and 6.4.7, we used random-number generation to simulate the roll of a six-sided die. In this next raylib case study exercise, you'll use dynamic visualization to make the Law of Large Numbers^{7,8} "come alive" in a die-rolling simulation that displays an animated bar chart. As the simulation repeatedly rolls the die, it updates an array of the frequencies with which each face appears. Then, it displays each die face's frequency, its percentage of the total rolls and a bar whose length represents the frequency's magnitude.

For a six-sided die, the face values 1 through 6 each should occur with "equal like-lihood"—the probability of each face appearing on any roll is $1/6^{th}$ or approximately 16.67%. If we roll a die 6000 times, we'd expect about 1000 of each face to appear. Like coin tossing, die rolling is random, so faces could occur fewer or more than 1000 times. As the number of die rolls increases, the Law of Large Numbers says that each of the frequencies should approach the expected value of 16.67%. If so, the bars in the bar chart should become nearly identical in length, as shown in the following screen captures of three sample executions for 60, 6000 and 6,000,0000 dice:



^{7. &}quot;Law of large numbers." Accessed December 18, 2020. https://encyclopediaofmath.org/index.php?title=Law_of_large_numbers.

^{8. &}quot;Law of large numbers." Accessed December 18, 2020. https://en.wikipedia.org/wiki/Law_of_large_numbers.





Running the Simulation on MacOs or Linux

When you execute this simulation, it requires two command-line arguments representing:

- the length of the simulation in animation frames, and
- the number of dice to roll per animation frame.

If the name of the program's executable is RollDieDynamic, the following macOS or Linux command will run the simulation for 60 animation frames, rolling one die per frame for a total of 60 rolls:

./RollDieDynamic 60 1

Similarly, the following will run the simulation for 600 animation frames, rolling 1000 dice per frame for a total of 600,000 rolls:

./RollDieDynamic 600 1000

Though we do not discuss the details of command-line arguments until Section 15.3, this completely coded simulation provides the statements you need to receive the command-line arguments.

Running the Simulation on MacOs or Linux

For the raylib self-contained Windows environment, perform the following steps to run this simulation:

- Open RollDieDynamic.c in Notepad++.
- **2.** Press the *F6* key.
- **3.** In the Execute dialog, modify the last line of the compilation and execution commands to include your command-line arguments, as in:

```
cmd /c IF EXIST $(NAME_PART).exe $(NAME_PART).exe 600 1000
```

Notepad++ replaces \$(NAME_PART) with the base name of the file you're running—RollDieDynamic in this case.

4. Click **OK** to compile and run the program.

Run the Program Several Times

Compile the simulation and run it several times, varying the command-line arguments. Next, study the simulation code (including extensive comments). As in our raylib games, you may pause the simulation at any time by pressing the *P* key and resume the simulation by pressing *P* again. Once you've studied the code, try Exercises 10.28–10.30, where you'll create visualizations for coin tossing, rolling two six-sided dice (producing the sums 2–12) and showing the win/loss results for the casino dice game Craps, based on the lengths of the games. You may want to use the techniques you've learned to analyze the results of playing popular card games like blackjack and various versions of poker.

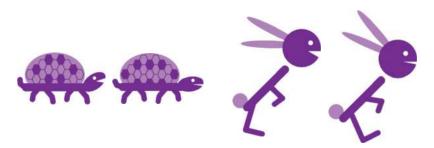
Case Study: The Tortoise and the Hare with raylib—a Multimedia "Extravaganza"

10.22 (Multimedia Tortoise and the Hare Race with Raylib) In this exercise, you'll use raylib graphics, animation and sound features from Exercises 10.19–10.21 to enhance Exercise 5.54's The Tortoise and the Hare Race. You'll incorporate a traditional horse race's sounds and multiple tortoise and hare images to create a fun, animated multimedia "extravaganza." For use in your race, we've provided in this chapter's examples folder a resources subfolder containing the following audio clips and images: 9

- An audio recording we created of the "Call to Post" trumpet piece played at the beginning of a horse race.
- A cannon-firing sound we created for our raylib Cannon game. You could use raylib's rFXGen sound generator (https://raylibtech.itch.io/rfxgen) to create a firing sound of your own.

^{9.} We created these audios and images. If you prefer, you could search the web for others or create your own. Be sure to comply with the license terms for any media you'll use in your applications.

- An audio clip we created of an announcer saying, "And they're off!" You could record yourself as the announcer saying this and other phrases to play throughout the race, such as "Tortoise pulls ahead!", "Hare pulls ahead!", "Down the stretch they come!", etc.
- A public-domain Wikimedia audio recording of the William Tell Overture¹⁰, which we have edited down to just the gallop portion (bada bum, bada bum, bada bum bum bum...) and placed in this chapter's resources folder for you to play during the race.
- Two slightly different tortoise images and two slightly different hare images we created:



We toggle between these images to create simple animations of the animals running. You can see the animations by viewing the tortoise.gif and hare.gif animated GIF images provided in the resources folder with this chapter's examples. Feel free to use these images or have some fun creating your own.

Implementing the Race

Implement the race using the basic raylib game structure you learned in the preceding raylib exercises. In your race, perform the following tasks:

- a) Before the race begins, play the trumpet audio of the "Call to Post," signifying that the racers should take their mark. As the "Call to Post" plays, the tortoise and hare should appear from the screen's left side and take their positions.
- b) Play the cannon sound to start the race, followed by the announcer saying, "And they're off!" At this point, the race animation begins.
- c) Throughout the race, play the provided William Tell Overture's gallop portion in the background repeatedly. See the raylib code sample at https://www.raylib.com/examples/web/audio/loader.html?name=audio_mu-sic_stream to learn how to play music in the background.
- d) As the tortoise and the hare move across the screen, toggle between each animal's two images to make it appear to be running. The tortoise moves slow-

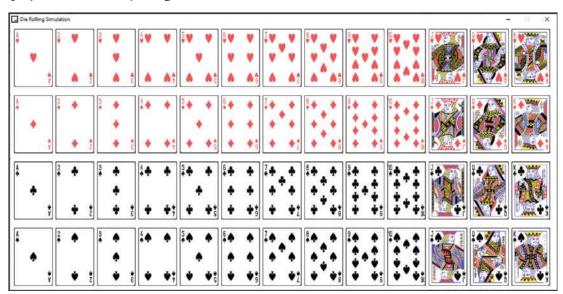
^{10. &}quot;File:Gioachino Rossini, William Tell Overture (military band version, 2000).ogg." Accessed January 2, 2021. https://commons.wikimedia.org/wiki/File:Gioachino_Rossini,_William_Tell_Overture_(military_band_version,_2000).ogg.

- er than the hare, so you may want to toggle between the tortoise's two images slower than between the hare's two images. When the hare sleeps, stop toggling between its images.
- e) When the tortoise and the hare are at the same position, display "OUCH!" for the turtle biting the hare and optionally play a high-pitched "OUCH!" audio clip.
- f) If the tortoise wins, display "Tortoise wins!" and optionally play a "Tortoise wins!" audio clip followed by cheering. If the hare wins, display "Hare wins" and optionally play a "Hare wins" audio clip followed by booing. If the race ends in a tie, you may want to favor the tortoise as the underdog or have the announcer say, "It's a tie!"
- g) You could play crowd cheering and booing sounds and additional announcer commentary as appropriate throughout the race. You might be able to find public-domain crowd sounds online.

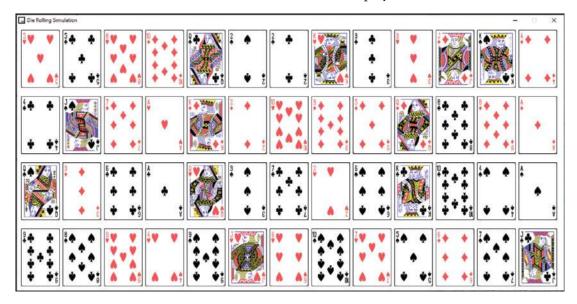
Random-Number Simulation Case Study: High-Performance Card Shuffling and Dealing with Card Images and raylib

10.23 (Card Shuffling and Dealing with Card Images and Raylib) In Section 10.7, you implemented a high-performance card-shuffling-and-dealing simulation using an array of Card objects. In this exercise, you'll incorporate raylib capabilities into your simulation and use them to display attractive, free public-domain card images for each card in the deck. Once you complete this exercise, you'll have the fundamental capabilities you need to begin implementing your favorite card games and to upgrade your solutions to the card-game exercises in earlier chapters.

You'll load the unshuffled 52 card images as raylib Texture2D objects, then display them in a 4-by-13 grid:



Each time the user clicks the mouse, shuffle and redisplay the cards:



Public-Domain Card Images from Wikimedia Commons

We downloaded these public-domain¹¹ card images from:

https://commons.wikimedia.org/wiki/ Category:SVG_English_pattern_playing_cards

and provided them for you in the card_images subfolder with this chapter's examples. We named each card-image file using the card's face and suit. For example, the images for the Spades suit are named as follows:

- Ace_of_Spades.png
- Deuce_of_Spades.png
- 3_of_Spades.png
- ...
- Jack_of_Spades.png
- Queen_of_Spades.png
- King_of_Spades.png

Implementing the Simulation

Using the basic raylib game framework you learned in Exercises 10.19–10.21, and the image-processing techniques you learned in Exercise 10.21, perform the following tasks:

- a) Modify Fig. 10.2's struct card definition to include a Texture2D member named image. This will store raylib's information about a loaded card image.
- b) When the application begins executing, initialize the unshuffled deck of cards in your raylib application's InitSimulation function. Modify the code that initializes your deck array to load each card's image. You can use string-

^{11.} https://creativecommons.org/publicdomain/zero/1.0/deed.en.

processing capabilities to assemble each card's *face* and *suit* strings into the card's image file name using the format:

```
face_of_suit.png (where you fill in the face and suit)
```

- c) The first time the raylib application's DrawFrame function executes, display the unshuffled array of 52 Card objects, as shown earlier. You'll need to perform calculations that determine each image's upper-left corner *x-y* coordinates.
- d) In the raylib application's Update function, check whether the user clicked the left mouse button. If so, shuffle the cards. The next call to DrawFrame will display the shuffled deck.

Drawing Notes

The following notes will help you implement your simulation:

- For this exercise, set the raylib window's screenWidth to 1280 and screen-Height to 620 to provide additional room for displaying the card images.
- When drawing each image, set the raylib function DrawTextureEx's scale argument to 0.25. This scales down the images, giving you enough room to draw them in four rows of 13 images each without overlapping one another.
- After drawing each image, use raylib function DrawRectangleLines as shown below to place a black border around each image for contrast with the window's white background:

```
DrawRectangleLines(x, y, deck[i].image.width * scale,
   deck[i].image.height * scale, BLACK);
```

Set variable scale to 0.25 to draw the rectangle using the same scale as the image.

Additional Raylib Exercises

10.24 (*Raylib Demos*) Compile, run and interact with several of raylib's bundled examples located in the raylib folder's examples subfolder. Study the source code provided for each of those examples to learn more about raylib's features.

10.25 (*Raylib Sample Games*) Compile, run and interact with several of raylib's sample games located in the raylib folder's games subfolder. Study the source code provided for each of those examples to learn more about raylib features. Be creative. Try modifying the games with your own enhancements.

10.26 (*Project: Enhanced SpotOn Game*) Try several of the following **SpotOn** game modifications:

- a) Display more spots for higher levels.
- b) Use bigger, possibly randomized speed boosts.
- c) Give a bonus for destroying multiple spots with one click.
- d) Use different point values for each spot color.

- e) Make the spots more elusive by allowing them to blink on and off, change direction spontaneously, change size spontaneously and move along nonlinear paths.
- f) Intermix smaller, harder-to-click spots.
- g) When the user clicks a spot, animate its destruction. For example, it could become concentric circles that fade away from the outside in, or it could become four pizza slices that spread out from the spot's center and fade away.
- h) Play a siren sound when the game moves to the next level.
- i) Display text for significant events like gaining or losing a life. The text can remain on the screen for a short time, then fade away.
- j) Have a specially colored, fast-moving spot. Clicking that spot destroys all the spots on the screen and gives the player a large point bonus.

10.27 (*Project: Enhanced Cannon Game*) Try several of the following Cannon game modifications:

- a) Display a dashed line showing the cannonball's path.
- b) Play a sound when the blocker hits the top or bottom of the screen.
- c) Play a sound when a target hits the top or bottom of the screen.
- d) Enhance the game to have levels. In each level, increase the number of target pieces.
- e) Keep score. Increase the user's score for each target piece hit by 10 times the current level. Decrease the score by 15 times the current level each time the cannonball hits the blocker. Display the score on the screen in the upper-left corner.
- f) Add cannonball and target explosion animations each time a cannonball hits a target.
- g) Add an explosion animation for the cannonball each time one hits the blocker.
- h) When the cannonball hits the blocker, increase the blocker's length by 5%.
- i) Make the game more difficult as it progresses by gradually increasing the speed of the targets and the blocker.
- j) Increase the number of independently moving blockers between the cannon and the targets.
- k) Add a bonus round that lasts for four seconds. Change the color of the targets and add music to indicate that it is a bonus round. If the user hits a target during those four seconds, give the user 1000 bonus points.
- l) Allow the user to move the cannon up and down via the arrow keys so it can be fired from different positions.

10.28 (Intro to Data Science: Dynamic Visualization of Coin Tossing) Modify the Law of Large Numbers die-rolling simulation from Exercise 10.21 to simulate flipping a coin. Use randomly generated 1s and 2s to represent heads and tails, respectively. Run simulations for 20, 200, 20,000 and 2,000,000 coin flips. Do you get approximately 50% heads and 50% tails? Do you see the "Law of Large Numbers" in action?

10.29 (Intro to Data Science: Dynamic Visualization of Rolling Two Dice) Modify our Law of Large Numbers die-rolling simulation from Exercise 10.21 to simulate rolling two dice. Calculate the sum of the two face values. Each die has a value from 1 to 6, so the sum will vary from 2 to 12, with 7 being the most frequent sum, and 2 and 12 the least frequent. The following diagram shows the 36 equally likely possible combinations of the two dice and their corresponding sums:

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

If you roll the dice 36,000 times:

- The values 2 and 12 each occur with a probability of 1/36th (2.778%), so you should expect about 1000 of each.
- The values 3 and 11 each occur with a probability of 2/36ths (5.556%), so you should expect about 2000 of each, and so on. You should expect about 6000 7s.

Display a dynamic bar plot with bars for each of the sums 2–12 summarizing their frequencies. Run the simulation for 360, 36,000 and 36,000,000 rolls.

10.30 (Intro to Data Science Project: Dynamic Visualization of Casino Game Win/Loss Statistics) Reimplement your solution to Exercise 6.20 using raylib to create a dynamic bar chart showing the wins and losses on the first roll, second roll, third roll, etc. Use pairs of green and red bars to indicate wins and losses, respectively, for each number of rolls.

10.31 (*Project: Brick Game*) Create a game similar to the cannon game that shoots pellets at a stationary brick wall. The goal is to destroy enough of the wall to shoot the moving target behind it. The faster you break through the wall and hit the target, the higher your score. Include multiple layers to the wall and a small moving target. Keep score. Increase difficulty with each round by building the wall using more layers and smaller bricks and increasing the speed of the moving target.

10.32 (*Project: Digital Clock*) Create an app that displays a digital clock on the screen.

10.33 (*Project: Analog Clock*) Create an app that displays an analog clock with hour, minute and second hands of appropriate lengths and thicknesses that rotate as the time changes.

File Processing



Objectives

In this chapter, you'll:

- Understand the concepts of files and streams.
- Write data to and read data from files using sequentialaccess text-file processing.
- Write data to, update data in and read data from files using random-access file processing and binary files.
- Develop a substantial transaction-processing program.
- Study Secure C programming in the context of file processing.

- **II.I** Introduction
- **11.2** Files and Streams
- 11.3 Creating a Sequential-Access File
 - 11.3.1 Pointer to a FILE
 - 11.3.2 Using fopen to Open a File
 - 11.3.3 Using feof to Check for the End-of-File Indicator
 - 11.3.4 Using fprintf to Write to a File
 - 11.3.5 Using fclose to Close a File
 - 11.3.6 File-Open Modes
- **11.4** Reading Data from a Sequential-Access File
 - 11.4.1 Resetting the File Position Pointer
 - 11.4.2 Credit Inquiry Program

- 11.5 Random-Access Files
- 11.6 Creating a Random-Access File
- 11.7 Writing Data Randomly to a Random-Access File
 - 11.7.1 Positioning the File Position Pointer with fseek
 - 11.7.2 Error Checking
- 11.8 Reading Data from a Random-Access File
- **11.9** Case Study: Transaction-Processing System
- **11.10** Secure C Programming

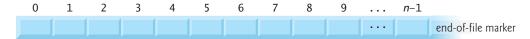
Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises |
All Case Study: Intro to NLP—Who Wrote Shakespeare's Works? |
All/Data-Science Case Study—Machine Learning with GNU Scientific Library |
All/Data-Science Case Study: Time Series and Simple Linear Regression |
Web Services and the Cloud Case Study—libcurl and OpenWeatherMap

11.1 Introduction

You studied the *data hierarchy* in Chapter 1. Data in variables is *temporary*—it's *lost* when a program terminates. Files enable long-term data retention. Computers store files on secondary storage devices, such as solid-state drives, flash drives and hard drives. This chapter explains how to create, update and process data files. We consider both sequential-access and random-access file processing.

11.2 Files and Streams

C views each file as a sequential stream of bytes, as shown in the following diagram:



Each file ends with an end-of-file marker or at a specific byte number recorded in a system-maintained, administrative data structure. This is platform-dependent and hidden from you.

Standard Streams in Every Program

When you open a file, C associates a stream with it. When program execution begins, C opens three streams automatically:

- The standard input stream receives input from the keyboard.
- The standard output stream displays output on the screen.
- The standard error stream displays error messages on the screen.

FILE Structure

Opening a file returns a pointer to a FILE structure (defined in <stdio.h>) containing information the program needs to process the file. In some operating systems, this structure includes a file descriptor—an integer index into an operating-system array called the open file table. Each array element contains a file control block (FCB)—information that the operating system uses to administer a particular file. You manipulate the standard input, standard output and standard error streams using the FILE pointers stdin, stdout and stderr.

File-Processing Function fgetc

The standard library provides many functions for reading data from and writing data to files. Function **fgetc**, like getchar, reads one character from the file specified by its FILE pointer argument. For example, the call fgetc(stdin) reads one character from the standard input stream. This call is equivalent to the call getchar().

File-Processing Function fputc

Function **fputc**, like **putchar**, writes the character in its first argument to the file specified by the FILE pointer in its second argument. For example, the function call fputc('a', stdout) writes a character to the standard output stream and is equivalent to putchar('a').

Other File-Processing Functions

Several other functions used to read data from standard input and write data to standard output have similarly named file-processing functions. The **fgets** and **fputs** functions, for example, read a line of text from a file and write a line of text to a file, respectively. The next few sections introduce the file-processing equivalents of functions scanf and printf—**fscanf** and **fprintf**. Later in the chapter, we discuss functions **fread** and **fwrite**.

✓ Self Check

I (Fill-In) When program execution begins, C opens three streams automatical
the,andstreams.
Answer: standard input, standard output, standard error.
2 (Fill-In) C views each file as a sequential stream of bytes. Each file ends eith
with a(n) or at a specific byte number recorded in a system-maintaine
administrative data structure.
Answer: end-of-file marker.

11.3 Creating a Sequential-Access File

C imposes no structure on a file. Thus, notions such as a record of a file are not part of the C language. The following example shows how you can impose your own record structure on a file.

Figure 11.1 creates a simple sequential-access file that might be used in an accounts-receivable system to track the amounts owed by a company's credit clients. For each client, the program obtains the client's account number, name and balance—the amount the client owes the company for goods and services received in the past. The data for each client constitutes a "record" for that client. The account number is this application's record key. This program assumes the user enters the records in account-number order. In a comprehensive accounts-receivable system, a sorting capability would enable the user to enter records in any order. The program would then sort the records and write them to the file. Figures 11.2–11.3 use the data file created in Fig. 11.1, so you must run the program in Fig. 11.1 before the programs in Figs. 11.2–11.3.

```
// fig11_01.c
 1
   // Creating a sequential file
   #include <stdio.h>
 4
    int main(void){
 5
 6
       FILE *cfPtr = NULL; // cfPtr = clients.txt file pointer
 7
 8
       // fopen opens the file. Exit the program if unable to create the file
       if ((cfPtr = fopen("clients.txt", "w")) == NULL) {
 9
          puts("File could not be opened");
10
       }
11
12
       else {
          puts("Enter the account, name, and balance.");
13
          puts("Enter EOF to end input.");
14
          printf("%s", "? ");
15
16
17
          int account = 0; // account number
          char name[30] = ""; // account name
18
19
          double balance = 0.0; // account balance
20
21
          scanf("%d%29s%1f", &account, name, &balance);
22
23
           // write account, name and balance into file with fprintf
24
          while (!feof(stdin)) {
             fprintf(cfPtr, "%d %s %.2f\n", account, name, balance);
printf("%s", "? ");
25
26
              scanf("%d%29s%1f", &account, name, &balance);
27
           }
28
29
          fclose(cfPtr); // fclose closes file
30
31
       }
    }
32
```

Fig. 11.1 | Creating a sequential file. (Part 1 of 2.)

```
Enter the account, name, and balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? \Z
```

Fig. 11.1 | Creating a sequential file. (Part 2 of 2.)

11.3.1 Pointer to a FILE

Line 6 defines cfPtr as a pointer to a FILE structure. A program refers to each open file with a separate FILE pointer. You need not know the FILE structure's specifics to use files. If you're interested, you can study its declaration in stdio.h.

11.3.2 Using fopen to Open a File

Line 9 calls fopen to create the file "clients.txt" and establish a "line of communication" with it. The file pointer that fopen returns is assigned to cfPtr.

Function fopen takes two arguments:

- a filename (which can include path information leading to the file's location) and
- a file open mode.

The file open mode "w" indicates fopen should open the file for writing. If the file does not exist and the file open mode is "w", fopen creates the file. If you open an existing file, fopen discards the file's contents without warning. This is a logic error, KERR if your program is not supposed to replace the existing file.



The if statement determines whether the file pointer cfPtr is NULL. If it's NULL, the file could not be opened, possibly because the program does not have permission to create a file in the specified folder. In this program, the file gets created in the same folder as the program. If cfPtr is NULL, the program prints an error message and terminates. Otherwise, the program processes the user's inputs and writes them to the file.

11.3.3 Using feof to Check for the End-of-File Indicator

The program prompts the user to enter the various fields for each record or to enter end-of-file when data entry is complete. The key combinations for end-of-file are platform-dependent:

- Windows: $\langle Ctrl \rangle + z$, then press *Enter*
- macOS/Linux: <*Ctrl>* + *d*

Line 24 calls **feof** to determine whether the end-of-file indicator is set for stdin. The end-of-file indicator informs the program that there's no more data to process. When the user enters the end-of-file key combination, the operating system sets the end-of-file indicator for the standard input stream. The feof function's argument is a FILE pointer to the file to test for the end-of-file indicator—stdin in this case. The function returns a nonzero (true) value when the end-of-file indicator has been set; otherwise, the function returns zero (false). This program's while statement continues executing until the user enters the end-of-file indicator.

11.3.4 Using fprintf to Write to a File

Line 25 writes a record as a line of text to the file clients.txt. You can retrieve the data later using a program designed to read the file (Section 11.4). The fprintf function is equivalent to printf, but fprintf also receives a FILE pointer argument specifying the file to which the data will be written. Function fprintf can output data to the standard output by using stdout as the FILE pointer argument.

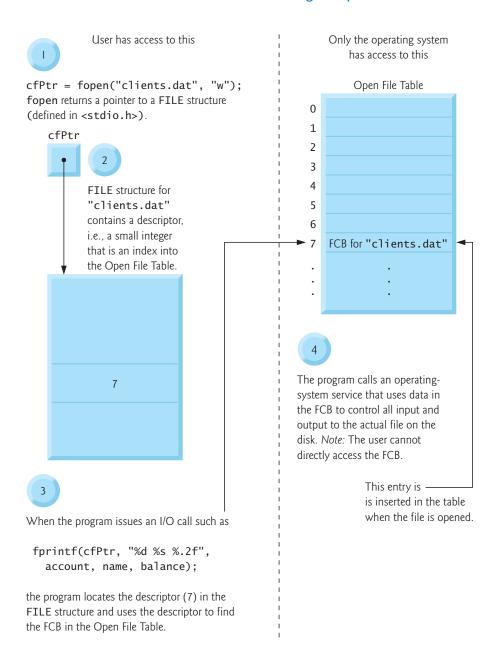
11.3.5 Using fclose to Close a File

After the user enters end-of-file, the program closes the clients.txt file by calling fclose (line 30), then terminates. Function fclose receives the FILE pointer as an argument. If you do not call fclose explicitly, the operating system normally will close the file when program execution terminates. This is an example of operatingsystem "housekeeping." You should close each file as soon as it's no longer needed. PERF : This frees resources for which other users or programs may be waiting.

In Fig. 11.1's sample execution, the user enters information for five accounts, then enters end-of-file to complete data entry. The sample execution does not show how the data records actually appear in the file. The next section presents a program that reads the file and displays its contents to verify that the program created the file successfully.

Relationship Between FILE Pointers, FILE Structures and FCBs

The following diagram illustrates the relationship between FILE pointers, FILE structures and FCBs. When a program opens "clients.txt", the operating system copies an FCB for the file into memory. The figure shows the connection between the file pointer returned by fopen and the FCB used by the operating system to administer the file. Programs may process no files, one file or several files. Each file has a different file pointer returned by fopen. All subsequent file-processing functions after the file is opened must refer to the file with the appropriate file pointer.



11.3.6 File-Open Modes

The following table summarizes the file-open modes. The ones containing the letter "b" are for manipulating binary files, which we discuss in Sections 11.5–11.9 when we introduce random-access files.

Mode	Description
r	Open an existing file for reading.
W	Create a file for writing. If the file already exists, discard the current contents.
a	Open or create a file for writing at the end of a file—this is for write operations that append data to a file.
r+	Open an existing file for update (reading and writing).
W+	Create a file for reading and writing. If the file already exists, discard the current contents.
a+	Open or create a file for reading and updating where all writing is done at the end of the file—that is, write operations append data to the file.
rb	Open an existing binary file for reading.
wb	Create a binary file for writing. If the file already exists, discard the current contents.
ab	Open or create a binary file for writing at the end of the file (appending).
rb+	Open an existing binary file for update (reading and writing).
wb+	Create a binary file for update. If the file already exists, discard the current contents.
ab+	Open or create a binary file for update. Writing is done at the end of the file.

CII Exclusive Write Mode

C11 added the exclusive write mode, ¹ indicated with "wx", "w+x", "wbx" or "wb+x". In exclusive write mode, fopen fails if the file already exists or cannot be created. If your program successfully opens a file in exclusive write mode and the underlying system supports exclusive file access, then *only* your program can access the file while it's open. If an error occurs while opening a file in any mode, fopen returns NULL.

ERR Common File-Processing Errors

Some common file-processing logic errors you might encounter include:

- Opening a nonexistent file for reading.
- Opening a file for reading or writing without having been granted the appropriate access rights to the file (this is operating-system dependent).
- Opening a file for writing when no space is available.
- Opening a file in write mode ("w") when it should be opened in update mode ("r+")—"w" discards the file's contents.

Self Check

Answer: The standard output device, which is normally the screen.

^{1.} Some compilers and platforms do not support exclusive write mode.

- **2** (*True/False*) The notion of a record of a file is built into C.
- **Answer:** False. Actually, C imposes no structure on a file, so notions such as a record of a file are not part of the C language. You can impose your own record structure on a file.
- **3** *(Fill-In)* A C program administers each file with a separate _____ structure. Answer: FILE.
- 4 (*True/False*) When you open a file for writing, fopen warns you if the file already exists.

Answer: False. fopen discards the file's contents without warning.

- 5 (Multiple Choice) Which file-open mode corresponds to the description, "open an existing file for update (reading and writing)"?
 - a) u.
 - b) rw.
 - c) r+.
 - d) w+.

Answer: c.

11.4 Reading Data from a Sequential-Access File

Data is stored in files so that it can be retrieved for processing when needed. The previous section demonstrated how to create a file for sequential access. This section shows how to read data sequentially from a file. If a file's contents should not be modified, open the file only for reading. This prevents unintentional modification of the file's contents and is another example of the principle of least privilege.

Figure 11.2 reads and displays records from the file "clients.txt" created in Fig. 11.1. Line 6 defines the FILE pointer cfPtr. Line 9 attempts to open the file for reading ("r") and determines whether it opened successfully—that is, fopen did not return NULL. Line 18 reads a "record" from the file. Function fscanf is equivalent to scanf but receives as its first argument a FILE pointer for the file from which to read. The first time this statement executes, account will have the value 100, name will have the value "Jones" and balance will have the value 24.98. Each subsequent call to fscanf (line 23) reads another record from the file and gives new values to account, name and balance. When there is no more data to read, line 26 closes the file, and the program terminates. Function feof returns true only after the program attempts to read past the file's last line.

```
// fig11_02.c
// Reading and printing a sequential file
#include <stdio.h>

int main(void) {
    FILE *cfPtr = NULL; // cfPtr = clients.txt file pointer
```

Fig. 11.2 | Reading and printing a sequential file. (Part 1 of 2.)

500

```
7
 8
        // fopen opens file; exits program if file cannot be opened
        if ((cfPtr = fopen("clients.txt", "r")) == NULL) {
 9
           puts("File could not be opened");
10
П
        else { // read account, name and balance from file
12
           int account = 0; // account number
13
           char name[30] = ""; // account name
14
15
           double balance = 0.0; // account balance
16
           printf("%-10s%-13s%s\n", "Account", "Name", "Balance");
17
           fscanf(cfPtr, "%d%29s%1f", &account, name, &balance);
18
19
20
           // while not end of file
21
           while (!feof(cfPtr)) {
              printf("%-10d%-13s%7.2f\n", account, name, balance);
fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);
22
23
           }
24
25
26
           fclose(cfPtr); // fclose closes the file
27
       }
   }
28
Account
                         Balance
           Name
100
           Jones
                           24.98
                          345.67
200
           Doe
300
           White
                            0.00
400
                          -42.16
           Stone
```

Fig. 11.2 | Reading and printing a sequential file. (Part 2 of 2.)

224.62

11.4.1 Resetting the File Position Pointer

When retrieving data sequentially from a file, a program normally reads from the beginning of the file until the desired data is found. In some cases, a program must process a file sequentially several times from the beginning. The statement

```
rewind(cfPtr);
```

Rich

repositions the file position pointer to the beginning (byte 0) of the file pointed to by cfPtr. The file position pointer is not really a pointer. It's an integer indicating the byte number of the next byte to read or write. This is sometimes referred to as the file offset. The file position pointer is a member of the FILE structure associated with each file.

11.4.2 Credit Inquiry Program

The program of Fig. 11.3 allows a credit manager to obtain lists of customers with:

- zero balances—customers who do not owe any money,
- credit balances—customers to whom the company owes money, or
- debit balances—customers who owe money for goods and services received.

A credit balance is a negative amount, and a debit balance is a positive amount. The program displays a menu and allows the credit manager to enter one of four options:

- Option 1 produces a list of accounts with zero balances.
- Option 2 produces a list of accounts with credit balances.
- Option 3 produces a list of accounts with debit balances.
- Option 4 terminates program execution.

```
I // fig11_03.c
 2 // Credit inquiry program
   #include <stdbool.h>
 4
   #include <stdio.h>
 5
 6
    enum Options {ZERO_BALANCE = 1, CREDIT_BALANCE, DEBIT_BALANCE, END};
 7
 8
    // determine whether to display a record
    bool shouldDisplay(enum Options option, double balance) {
 9
       if ((option == ZERO_BALANCE) && (balance == 0)) {
10
П
          return true;
       }
12
13
       if ((option == CREDIT_BALANCE) && (balance < 0)) {</pre>
14
15
          return true;
       }
16
17
18
       if ((option == DEBIT_BALANCE) && (balance > 0)) {
19
          return true;
20
       }
21
22
       return false;
23
    }
24
25
    int main(void) {
       FILE *cfPtr = NULL; // clients.txt file pointer
26
27
28
       // fopen opens the file; exits program if file cannot be opened
       if ((cfPtr = fopen("clients.txt", "r")) == NULL) {
29
          puts("File could not be opened");
30
31
       }
32
       else {
33
          // display request options
          printf("%s", "Enter request\n"
34
35
             " 1 - List accounts with zero balances\n"
             " 2 - List accounts with credit balances\n"
36
             " 3 - List accounts with debit balances\n"
37
             " 4 - End of run\n? ");
38
39
          int request = 0;
40
          scanf("%d", &request);
41
```

Fig. 11.3 | Credit inquiry program. (Part 1 of 3.)

```
42
          // display records
43
          while (request != END) {
44
             switch (request) {
                case ZERO_BALANCE:
45
46
                    puts("\nAccounts with zero balances:");
47
                    break:
48
                case CREDIT_BALANCE:
49
                    puts("\nAccounts with credit balances:");
50
                    break;
51
                case DEBIT_BALANCE:
52
                    puts("\nAccounts with debit balances:");
53
                    break;
54
             }
55
56
             int account = 0;
             char name[30] = "";
57
58
             double balance = 0.0;
59
60
             // read account, name and balance from file
             fscanf(cfPtr, "%d%29s%1f", &account, name, &balance);
61
62
63
             // read file contents (until eof)
             while (!feof(cfPtr)) {
64
65
                // output only if balance is 0
66
                if (shouldDisplay(request, balance)) {
67
                    printf("%-10d%-13s%7.2f\n", account, name, balance);
                }
68
69
                 // read account, name and balance from file
70
71
                fscanf(cfPtr, "%d%29s%1f", &account, name, &balance);
             }
72
73
74
             rewind(cfPtr); // return cfPtr to beginning of file
75
             printf("%s", "\n? ");
76
             scanf("%d", &request);
77
78
79
          puts("End of run.");
80
81
          fclose(cfPtr); // close the file
82
       }
83 }
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 1
Accounts with zero balances:
300
          White
                           0.00
```

Fig. 11.3 | Credit inquiry program. (Part 2 of 3.)

```
? 2
Accounts with credit balances:
400
          Stone
                        -42.16
? 3
Accounts with debit balances:
                         24.98
100
          Jones
200
                         345.67
          Doe
500
                         224.62
          Rich
End of run.
```

Fig. 11.3 | Credit inquiry program. (Part 3 of 3.)

Updating a Sequential File

You cannot modify data in this type of sequential file without the risk of destroying other data. For example, if the name "White" needs to be changed to "Worthington", you cannot simply overwrite the old name. The record for White was written to the file as

```
300 White 0.00
```

If you were to rewrite the record beginning at the same location in the file using the new name, the record would be

```
300 Worthington 0.00
```

The new record has more characters than the original record. The characters beyond the second "o" in "Worthington" will *overwrite* the beginning of the next sequential record in the file. The problem here is that in the **formatted input/output model** using fprintf and fscanf, fields and records can vary in size. For example, the values 7, 14, -117, 2074 and 27383 are all ints stored internally in the same number of bytes, but they're different-sized fields when displayed on the screen or written to a file as text.

So, sequential access with fprintf and fscanf typically is not used to update records in place. Instead, the entire file is rewritten. In a sequential-access file, we'd make the preceding name change by

- copying the records before 300 White 0.00 to a new file,
- writing the new record,
- copying the records after 300 White 0.00 to the new file, then
- replacing the old file with the new one.

This requires processing every record in the file to update one record.



(*Fill-In*) Function fscanf is equivalent to function scanf, but fscanf receives as an argument a(n) _____.

Answer: file pointer for the file from which to read data.

2 (*TruelFalse*) Function feof returns true only after the program attempts to read the nonexistent data following the last line.

Answer: True.

3 (Fill-In) The following statement repositions a file's _____ to the file's byte 0. rewind(cfPtr);

Answer: file position pointer.

4 (*TruelFalse*) In the formatted input/output model using fprintf and fscanf, fields—and hence records—are fixed in size.

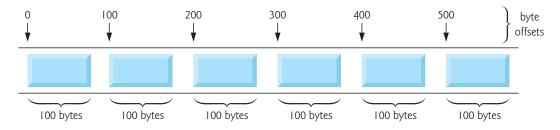
Answer: False. Actually, in this model, fields—and hence records—can vary in size.

11.5 Random-Access Files

Records that you create with the formatted output function fprintf may vary in length. A random-access file, on the other hand, uses *fixed-length* records that may be accessed directly (and thus quickly) without searching through other records. This makes random-access files appropriate for transaction-processing systems that require rapid access to specific data, such as airline reservation systems, banking systems and point-of-sale systems. There are other ways to implement random-access files, but we'll limit our discussion to this straightforward approach using fixed-length records.

Because every record in a random-access file normally has the same length, each record's exact location relative to the beginning of the file can be calculated as a function of the record key. We'll soon see how this facilitates immediate access to specific records, even in large files.

The following diagram illustrates one way to implement a random-access file. Such a file is like a freight train with many cars—some empty and some with cargo. Each car in the train has the same length.



Fixed-length records enable a program to insert data in a random-access file without destroying other data in the file. Data stored previously also can be updated or deleted without rewriting the entire file. In the sections that follow, we explain how to

- create a random-access file,
- enter data,
- read the data both sequentially and randomly,
- update the data, and
- delete data that's no longer needed.

Self Check

- I (*True/False*) Individual records that you write to and read from a random-access file may be accessed directly without searching through other records. This makes random-access files appropriate for systems that require rapid access to specific data. Answer: *True*.
- **2** (*Fill-In*) A random-access file uses fixed-length records, so the exact location of a record relative to the beginning of the file can be calculated based on the _____. Answer: record key.

11.6 Creating a Random-Access File

Function fwrite writes a specified number of bytes from a specified location in memory to a file. The data is written at the file position pointer's current location. Function fread reads a specified number of bytes from the file position pointer's current location to a specified area in memory. Writing a four-byte integer with

```
fprintf(fPtr, "%d", number);
```

could output as many as 11 digits—10 digits plus a sign, each of which requires at least one byte of storage, based on the character set for the locale. With random-access files, the statement

```
fwrite(&number, sizeof(int), 1, fPtr);
```

always writes four bytes (on a system with four-byte integers) from the int variable number to the file represented by fPtr. We'll explain the argument 1 in a moment. Later, we can use fread to read those four bytes into an int variable. Although fread and fwrite read and write data in fixed-size rather than variable-size format, they process data as "raw" bytes, rather than in printf's and scanf's human-readable text format. The "raw" data representation is system-dependent, so "raw" data may not be readable on other systems, or by programs produced by other compilers or with different compilation options.

fwrite and fread Can Write and Read Arrays

Functions fwrite and fread can write and read arrays. The third argument of both fwrite and fread is the number of elements to write or read. The preceding fwrite function call writes a single integer to a file, so the third argument is 1—as if we were writing one array element. File-processing programs rarely write a single field to a file. Normally, they write one struct at a time, as we show in the following examples.

Problem Statement

Consider the following problem statement:

Create a transaction-processing system capable of storing up to 100 fixed-length records. Each record should have an account number (the record key), a last name, a first name and a balance. The program should use a random-access file and should be able to update an account, insert a new account, delete an account and list all the records in a formatted text file for printing.

The next several sections introduce the techniques we'll use to create the transaction-processing program. Figure 11.4 shows how to open a random-access file, define a record format using a struct, write data to the file and close the file. This program initializes all 100 records of the file "accounts.dat" with empty structs using the function fwrite. Each empty struct contains the account number 0, empty strings ("") for the last and first names, and the balance 0.0. We initialize all the records to create the space in which the file will be stored and to make it possible to determine whether a record contains data.

```
// fig11_04.c
   // Creating a random-access file sequentially
 2
 3
   #include <stdio.h>
 5 // clientData structure definition
 6
    struct clientData {
 7
       int account:
 8
       char lastName[15];
 9
       char firstName[10];
10
       double balance;
    };
11
12
13
    int main(void) {
       FILE *cfPtr = NULL; // accounts.dat file pointer
14
15
       // fopen opens the file; exits if file cannot be opened
16
       if ((cfPtr = fopen("accounts.dat", "wb")) == NULL) {
17
          puts("File could not be opened.");
18
19
       }
       else {
20
          // create clientData with default information
21
          struct clientData blankClient = {0, "", "", 0.0};
22
23
24
          // output 100 blank records to file
25
          for (int i = 1; i <= 100; ++i) {
             fwrite(&blankClient, sizeof(struct clientData), 1, cfPtr);
26
27
28
          fclose (cfPtr); // fclose closes the file
29
       }
30
    }
31
```

Fig. 11.4 | Creating a random-access file sequentially.

Line 17 opens the file "accounts.dat" for writing in binary mode ("wb"). Function fwrite (line 26) writes a block of bytes to a file. The arguments are:

- &blankClient—the address of the object to write,
- sizeof(struct clientData)—the size in bytes of the object to write,
- 1—the number of objects of that size to write, and
- cfPtr—a FILE * representing the file in which the bytes will be stored.

Recall that sizeof returns the size in bytes of its operand, struct clientData.

Writing an Array of Objects

In line 26, fwrite writes one object that's not an array element. To write an array, pass it to fwrite as the first argument and specify as the third argument the number of elements to output.

✓ Self Check

I (*TruelFalse*) For a four-byte int variable number, the following statement always writes four bytes, even if the number's text representation could be as many as 11 digits:

```
fwrite(&number, sizeof(int), 1, fPtr);
```

Answer: True.

2 (*Fill-In*) Functions fread and fwrite read and write data in "raw data" format—that is, as ______ of data.

Answer: bytes.

3 *(Fill-In)* Function fwrite can write several array elements. In the call to fwrite, specify a pointer to an array and _____ as the first and third arguments.

Answer: the number of elements to write.

11.7 Writing Data Randomly to a Random-Access File

[Note: Figures 11.5, 11.6 and 11.7 use the data file created in Fig. 11.4, so you must run Fig. 11.4 before Figs. 11.5, 11.6 and 11.7.]

Figure 11.5 writes data to the file "accounts.dat". It uses **fseek** and fwrite to store data at specific locations in the file. Function fseek sets the file position pointer to a specific byte position, then fwrite writes the data there.

```
I // fig11_05.c
2 // Writing data randomly to a random-access file
3 #include <stdio.h>
```

Fig. 11.5 Writing data randomly to a random-access file. (Part 1 of 3.)

5

```
// clientData structure definition
   struct clientData {
 7
       int account:
       char lastName[15];
 8
 9
       char firstName[10];
10
       double balance;
    }: // end structure clientData
11
12
13
    int main(void) {
       FILE *cfPtr = NULL; // accounts.dat file pointer
14
15
16
       // fopen opens the file; exits if file cannot be opened
       if ((cfPtr = fopen("accounts.dat", "rb+")) == NULL) {
17
18
          puts("File could not be opened.");
19
       }
20
       else {
          // create clientData with default information
21
          struct clientData client = {0, "", "", 0.0};
22
23
          // require user to specify account number
24
          printf("%s", "Enter account number (1 to 100, 0 to end input): ");
25
          scanf("%d", &client.account);
26
27
          // user enters information, which is copied into file
28
29
          while (client.account != 0) {
             // user enters last name, first name and balance
30
             printf("%s", "Enter lastname, firstname, balance: ");
31
32
33
             // set record lastName, firstName and balance value
34
             fscanf(stdin, "%14s%9s%1f", client.lastName,
                client.firstName, &client.balance);
35
36
             // seek position in file to user-specified record
37
38
             fseek(cfPtr, (client.account - 1) *
                sizeof(struct clientData), SEEK_SET);
39
40
41
             // write user-specified information in file
42
             fwrite(&client, sizeof(struct clientData), 1, cfPtr);
43
44
             // enable user to input another account number
             printf("%s", "\nEnter account number: ");
45
46
             scanf("%d", &client.account);
47
48
49
          fclose(cfPtr); // fclose closes the file
50
       }
51
   }
Enter account number (1 to 100, 0 to end input): 37
Enter lastname, firstname, balance: Barker Doug 0.00
Enter account number: 29
Enter lastname, firstname, balance: Brown Nancy -24.54
```

Fig. 11.5 Writing data randomly to a random-access file. (Part 2 of 3.)

```
Enter account number: 96
Enter lastname, firstname, balance: Stone Sam 34.98
Enter account number: 88
Enter lastname, firstname, balance: Smith Dave 258.34
Enter account number: 33
Enter lastname, firstname, balance: Dunn Stacey 314.33
Enter account number: 0
```

Fig. 11.5 Writing data randomly to a random-access file. (Part 3 of 3.)

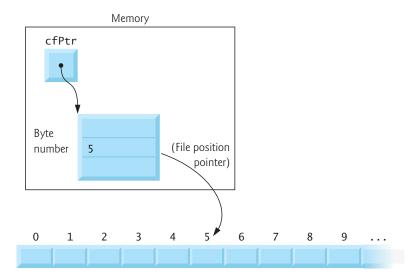
11.7.1 Positioning the File Position Pointer with fseek

Lines 38–39 position the file position pointer for the file referenced by cfPtr to the byte location calculated by

```
(client.account - 1) * sizeof(struct clientData)
```

This expression's value is the offset or displacement. In this example, the account number is 1–100. The file starts with byte 0, so we subtract 1 from the account number when calculating the record's byte location. For record 1, lines 38–39 set the file position pointer to byte 0 of the file. The symbolic constant SEEK_SET indicates that fseek should move the file position pointer relative to the beginning of the file.

The following diagram illustrates the FILE pointer referring to a FILE structure in memory. The file position pointer in this diagram indicates that the next byte to be read or written is byte number 5.



fseek Function Prototype

The function prototype for fseek is

```
int fseek(FILE *stream, long int offset, int whence);
```

where offset is the number of bytes to seek from whence in the file pointed to by stream. Positive offsets seek forward, and negative offsets seek backward. The argument whence can be SEEK_SET, SEEK_CUR or SEEK_END (all defined in <stdio.h>), which indicate the location from which the seek begins:

- SEEK_SET indicates that the seek is measured from the *beginning* of the file.
- SEEK_CUR indicates that the seek is measured from the *current location* in the file.
- SEEK_END indicates that the seek is measured from the *end* of the file.

You should use only positive offsets with SEEK_SET and only negative ones with SEEK_END.

11.7.2 Error Checking

For simplicity, the programs in this chapter do *not* perform error checking. Industrial-strength programs should determine whether functions such as fscanf (Fig. 11.5, lines 34–35), fseek (lines 38–39) and fwrite (line 42) operate correctly by checking their return values. Function fscanf returns the number of data items successfully read or the value EOF if a problem occurs while reading data. Function fseek returns a nonzero value if the seek operation cannot be performed (e.g., attempting to seek to a position before the start of the file). Function fwrite returns the number of items it successfully output. If this number is less than the *third argument* in the function call, then a write error occurred.

✓ Self Check

[(Fill-In) Function _____ sets the file position pointer to a specific byte position in the file.

Answer: fseek.

2 (Fill-In) The symbolic constant _____ indicates that the file position pointer should be positioned relative to the beginning of the file.

Answer: SEEK_SET.

11.8 Reading Data from a Random-Access File

Function fread reads a specified number of bytes from a file into memory. For example, fread(&client, sizeof(struct clientData), 1, cfPtr);

reads the number of bytes determined by sizeof(struct clientData) from the file referenced by cfPtr, stores the data in client and returns the number of bytes read. It reads bytes from the location specified by the file position pointer.

Function fread can read several fixed-size array elements by providing a pointer to the array in which the elements will be stored and by indicating the number of elements to be read. The preceding statement reads one element. To read more than one, specify the number of elements as fread's third argument. Function fread returns the number of items it successfully input. If this number is less than the function call's third argument, a read error occurred.

Figure 11.6 sequentially reads each record in the "accounts.dat" file, determines whether it contains data and, if so, displays the formatted data. Function feof determines when the end of the file is reached, and the fread function (lines 28–29) transfers data from the file to the clientData structure client.

```
// fig11_06.c
 1
   // Reading a random-access file sequentially
 2
   #include <stdio.h>
 4
 5
    // clientData structure definition
    struct clientData {
 7
       int account;
 8
       char lastName[15];
 9
       char firstName[10];
10
       double balance;
11
    };
12
13
    int main(void){
       FILE *cfPtr = NULL; // accounts.dat file pointer
14
15
16
       // fopen opens the file; exits if file cannot be opened
       if ((cfPtr = fopen("accounts.dat", "rb")) == NULL) {
17
18
          puts("File could not be opened.");
19
       }
       else {
20
          printf("%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
21
             "First Name", "Balance");
22
23
          // read all records from file (until eof)
24
          while (!feof(cfPtr)) {
25
26
             // read a record
             struct clientData client = {0, "", "", 0.0};
27
28
             size_t result =
                 fread(&client, sizeof(struct clientData), 1, cfPtr);
29
30
31
             // display record
             if (result != 0 && client.account != 0) {
32
                printf("%-6d%-16s%-11s%10.2f\n", client.account,
33
34
                    client.lastName, client.firstName, client.balance);
35
             }
          }
36
37
38
          fclose(cfPtr); // fclose closes the file
39
       }
    }
40
```

Fig. 11.6 Reading a random-access file sequentially. (Part 1 of 2.)

|--|

Fig. 11.6 Reading a random-access file sequentially. (Part 2 of 2.)



Self Check

- I (*True/False*) Function fread returns the number of bytes it successfully input. Answer: *False*. Function fread returns the number of *items* it successfully input. Each item can be many bytes. If the number of items is fewer than fread's third argument, then the read operation did not complete successfully.
- **2** (*True/False*) Function fread can read several fixed-size array elements by providing a pointer to the array in which the elements will be stored and by indicating the number of elements to be read.

Answer: True.

11.9 Case Study: Transaction-Processing System

Let's create a transaction-processing program (Fig. 11.7) using random-access files. The program maintains a bank's account information—updating existing accounts, adding new accounts, deleting accounts and storing a listing of current accounts in a text file for printing. We assume that the program of Fig. 11.4 created the file accounts.dat.

Option 1: Create a Formatted List of Accounts

The program has five options—option 5 terminates the program. Option 1 calls function textFile (lines 58–86) to store a formatted account report in a text file called accounts.txt, which can be printed later. The function uses fread and the sequential file-access techniques shown in Fig. 11.6. After option 1, accounts.txt contains:

Acct 29 33 37	Last Name Brown Dunn Barker	First Name Nancy Stacey Doug	Balance -24.54 314.33 0.00
88	Smith	Dave	258.34
96	Stone	Sam	34.98

Option 2: Update an Account

Option 2 calls the function updateRecord (lines 89–125) to update an account. The function updates only a record that already exists, so the function first checks whether the record specified by the user is empty. First, we read the record into structure client with fread. If the member account is 0, the record contains no information. So,

the program displays a message that the record is empty, then redisplays the menu choices. If the record contains information, function updateRecord inputs the transaction amount, calculates the new balance and rewrites the record to the file. A typical output for option 2 is

```
Enter account to update (1 - 100): 37
37 Barker Doug 0.00

Enter charge (+) or payment (-): +87.99
37 Barker Doug 87.99
```

Option 3: Create a New Account

Option 3 calls the function newRecord (lines 128–161) to add a new account to the file. If the user enters an account number for an existing account, newRecord displays an error message indicating that the record already contains information, and the menu choices are printed again. This function uses the same process to add a new account, as does the program in Fig. 11.5. A typical output for option 3 is

```
Enter new account number (1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

Option 4: Delete an Account

Option 4 calls function deleteRecord (lines 164–190) to delete a record from the file. Deletion is accomplished by asking the user for the account number and reinitializing the record. If the account contains no information, deleteRecord displays an error message indicating that the account does not exist.

Code for the Transaction-Processing Program

The program is shown in Fig. 11.7. The file "accounts.dat" is opened for update (reading and writing) using "rb+" mode.

```
// fig11_07.c
2 // Transaction-processing program reads a random-access file sequentially,
3 // updates data already written to the file, creates new data to
4 // be placed in the file, and deletes data previously stored in the file.
5 #include <stdio.h>
7 // clientData structure definition
8
   struct clientData {
9
      int account;
10
      char lastName[15];
11
      char firstName[10];
12
      double balance;
13 };
```

Fig. 11.7 Transaction-processing program. (Part 1 of 5.)

```
14
15
    // prototypes
int enterChoice(void);
17 void textFile(FILE *readPtr);
   void updateRecord(FILE *fPtr);
   void newRecord(FILE *fPtr);
19
   void deleteRecord(FILE *fPtr);
20
21
22
   int main(void) {
       FILE *cfPtr = NULL; // accounts.dat file pointer
23
24
25
       // fopen opens the file; exits if file cannot be opened
       if ((cfPtr = fopen("accounts.dat", "rb+")) == NULL) {
26
27
          puts("File could not be opened.");
28
       }
29
       else {
30
          int choice = 0; // user
31
          // enable user to specify action
32
          while ((choice = enterChoice()) != 5) {
33
34
             switch (choice) {
35
                case 1: // create text file from record file
36
                   textFile(cfPtr);
37
                   break;
                case 2: // update record
38
39
                   updateRecord(cfPtr);
40
                   break:
                case 3: // create record
41
42
                   newRecord(cfPtr);
43
                   break:
                case 4: // delete existing record
44
                   deleteRecord(cfPtr);
45
46
                   break;
47
                default: // display message for invalid choice
                   puts("Incorrect choice");
48
49
                   break;
50
             }
          }
51
52
53
          fclose(cfPtr); // fclose closes the file
54
       }
    }
55
56
57
    // create formatted text file for printing
58
    void textFile(FILE *readPtr) {
       FILE *writePtr = NULL; // accounts.txt file pointer
59
60
61
       // fopen opens the file; exits if file cannot be opened
62
       if ((writePtr = fopen("accounts.txt", "w")) == NULL) {
63
          puts("File could not be opened.");
       }
64
```

Fig. 11.7 | Transaction-processing program. (Part 2 of 5.)

```
65
       else {
66
          rewind(readPtr); // sets pointer to beginning of file
          fprintf(writePtr, "%-6s%-16s%-11s%10s\n",
67
68
              "Acct", "Last Name", "First Name", "Balance");
69
70
          // copy all records from random-access file into text file
          while (!feof(readPtr)) {
71
             // create clientData with default information
72
             struct clientData client = {0, "", "", 0.0};
73
74
             size_t result =
                fread(&client, sizeof(struct clientData), 1, readPtr);
75
76
77
             // write single record to text file
78
             if (result != 0 && client.account != 0) {
                fprintf(writePtr, "%-6d%-16s%-11s%10.2f\n", client.account,
79
                    client.lastName, client.firstName, client.balance);
80
81
             }
          }
82
83
84
          fclose(writePtr); // fclose closes the file
85
       }
    }
86
87
    // update balance in record
88
    void updateRecord(FILE *fPtr) {
89
       // obtain number of account to update
90
       printf("%s", "Enter account to update (1 - 100): ");
91
92
       int account = 0; // account number
       scanf("%d", &account);
93
94
95
       // move file pointer to correct record in file
96
       fseek(fPtr, (account - 1) * sizeof(struct clientData), SEEK_SET);
97
       // read record from file
98
       struct clientData client = {0, "", "", 0.0};
99
       fread(&client, sizeof(struct clientData), 1, fPtr);
100
101
       // display error if account does not exist
102
       if (client.account == 0) {
103
          printf("Account #%d has no information.\n", account);
104
105
       }
106
       else { // update record
          printf("%-6d%-16s%-11s%10.2f\n\n", client.account, client.lastName,
107
             client.firstName, client.balance);
108
109
110
          // request transaction amount from user
III
          printf("%s", "Enter charge (+) or payment (-): ");
          double transaction = 0.0; // transaction amount
112
113
          scanf("%1f", &transaction);
114
          client.balance += transaction; // update record balance
115
```

Fig. 11.7 Transaction-processing program. (Part 3 of 5.)

```
116
          printf("%-6d%-16s%-11s%10.2f\n", client.account, client.lastName,
117
             client.firstName, client.balance);
118
          // move file pointer to correct record in file
119
120
          fseek(fPtr, (account - 1) * sizeof(struct clientData), SEEK_SET);
121
          // write updated record over old record in file
122
123
          fwrite(&client, sizeof(struct clientData), 1, fPtr);
124
       }
125 }
126
127 // create and insert record
128 void newRecord(FILE *fPtr) {
129
       // obtain number of account to create
       printf("%s", "Enter new account number (1 - 100): ");
130
       int account = 0; // account number
131
       scanf("%d", &account);
132
133
       // move file pointer to correct record in file
134
135
       fseek(fPtr, (account - 1) * sizeof(struct clientData), SEEK_SET);
136
       // read record from file
137
       struct clientData client = {0, "", "", 0.0};
138
139
       fread(&client, sizeof(struct clientData), 1, fPtr);
140
       // display error if account already exists
141
142
       if (client.account != 0) {
          printf("Account #%d already contains information.\n",
143
144
             client.account);
145
       }
       else { // create record
146
          // user enters last name, first name and balance
147
          printf("%s", "Enter lastname, firstname, balance\n? ");
148
          scanf("%14s%9s%1f", &client.lastName, &client.firstName.
149
             &client.balance);
150
151
152
          client.account = account;
153
          // move file pointer to correct record in file
154
          fseek(fPtr, (client.account - 1) * sizeof(struct clientData),
155
156
             SEEK_SET);
157
          // insert record in file
158
          fwrite(&client, sizeof(struct clientData), 1, fPtr);
159
160
       }
161 }
162
163 // delete an existing record
164 void deleteRecord(FILE *fPtr) {
165
       // obtain number of account to delete
       printf("%s", "Enter account number to delete (1 - 100): ");
166
       int account = 0; // account number
167
168
       scanf("%d", &account);
```

Fig. 11.7 | Transaction-processing program. (Part 4 of 5.)

```
169
170
       // move file pointer to correct record in file
       fseek(fPtr, (account - 1) * sizeof(struct clientData), SEEK_SET);
171
172
       // read record from file
173
       struct clientData client = {0, "", "", 0.0};
174
       fread(&client, sizeof(struct clientData), 1, fPtr);
175
176
       // display error if record does not exist
177
       if (client.account == 0) {
178
          printf("Account %d does not exist.\n", account);
179
180
181
       else { // delete record
182
          // move file pointer to correct record in file
          fseek(fPtr, (account - 1) * sizeof(struct clientData), SEEK_SET);
183
184
          struct clientData blankClient = {0, "", "", 0}; // blank client
185
186
           // replace existing record with blank record
187
          fwrite(&blankClient, sizeof(struct clientData), 1, fPtr);
188
189
       }
190 }
191
192 // enable user to input menu choice
193 int enterChoice(void) {
       // display available options
194
       printf("%s", "\nEnter your choice\n"
195
           "1 - store a formatted text file of accounts called\n"
196
197
               \"accounts.txt\" for printing\n"
          "2 - update an account\n"
198
          "3 - add a new account\n"
199
          "4 - delete an account\n"
200
          "5 - end program\n? ");
201
202
       int menuChoice = 0; // variable to store user
203
       scanf("%d", &menuChoice); // receive choice from user
204
205
       return menuChoice;
206 }
```

Fig. 11.7 | Transaction-processing program. (Part 5 of 5.)

Related Exercises

This Transaction-Processing System case study is supported by Exercise 11.11 (Hardware Inventory) and Exercise 11.17 (Modified Transaction-Processing System).

Self Check

(Discussion) In the following code, what does the if statement's condition test?
if ((cfPtr = fopen("accounts.dat", "rb+")) == NULL)

Answer: The condition tests whether the file accounts.dat was opened successfully in binary mode for reading and writing.

2 (Discussion) What does the following statement do in the program of Fig. 11.7? fseek(fPtr, (account - 1) * sizeof(struct clientData), SEEK_SET);

Answer: This statement moves the file position pointer for the file that fPtr represents to the position for the clientData record account.

11.10 Secure C Programming

fprintf_s and fscanf_s

The examples in Sections 11.3–11.4 used functions fprintf and fscanf to write text to and read text from files, respectively. The C standard's Annex K provides versions of these functions named fprintf_s and fscanf_s that are identical to the printf_s and scanf_s functions we've previously introduced, except that you also specify a FILE pointer argument indicating the file to manipulate. If your C compiler's standard libraries include these functions, you should use them instead of fprintf and fscanf. As with scanf_s and printf_s, Microsoft's versions of fprintf_s and fscanf_s differ from those in Annex K.

Chapter 9 of the SEI CERT C Coding Standard

Chapter 9 of the SEI CERT C Coding Standard is dedicated to input/output recommendations and rules—many apply to file processing in general, and several of these apply to the file-processing functions presented in this chapter. For more information on each, visit https://wiki.sei.cmu.edu/:

- FIO03-C: When opening a file for writing using the nonexclusive file-open modes discussed in this chapter, if the file exists, function fopen opens it and truncates its contents, providing no indication of whether the file existed before the fopen call. To ensure that an existing file is *not* opened and truncated, you can use C11's *exclusive write mode* (discussed in Section 11.3), which allows fopen to open the file *only* if it does *not* already exist.
- FIO04-C: In industrial-strength code, you should always check the return values of file-processing functions that return error indicators to ensure that the functions performed their tasks correctly.
- FIO07-C. Function rewind does not return a value, so you cannot test whether the operation was successful. It's recommended instead that you use function fseek because it returns a nonzero value if it fails.
- FIO09-C. We demonstrated both text files and binary files in this chapter. Due to differences in binary data representations across platforms, files written in binary format often are *not* portable. For more portable file representations, consider using text files or a function library that can handle the differences in binary file representations across platforms.
- FIO14-C. Some library functions do not operate identically on text files and binary files. In particular, function fseek is *not* guaranteed to work correctly with binary files if you seek from SEEK_END, so SEEK_SET should be used.

FIO42-C. On many platforms, you can have only a limited number of files
open at once. For this reason, you should always close a file as soon as it's no
longer needed by your program.

✓ Self Check

I (*Fill-In*) When you open a file for writing, you can ensure that an existing file is not truncated by using ______, which allows fopen to open the file only if it does not exist.

Answer: exclusive write mode.

2 (*TruelFalse*) Function rewind does not return a value, so you cannot test whether the operation was successful. Instead, use function fseek because it returns a nonzero value if it fails.

Answer: True.

3 (*TruelFalse*) Many platforms allow only a limited number of files to be open at once. So, you should always close a file as soon as it's no longer needed.

Answer: True.

Summary

Section 11.1 Introduction

- Files (p. 540) are used for permanent retention of large amounts of data.
- Computers store files on **secondary storage devices**, such as solid-state drives, flash drives and hard drives.

Section 11.2 Files and Streams

- C views each file as a sequential **stream of bytes** (p. 540). When a file is opened, a stream is associated with the file.
- Three streams are automatically opened when program execution begins—the standard input (p. 540), the standard output (p. 540) and the standard error (p. 540).
- Streams provide communication channels between files and programs.
- The standard input stream enables a program to read data from the keyboard, and the standard output stream enables a program to print data on the screen.
- Opening a file returns a pointer to a **FILE structure** (defined in <stdio.h>; p. 541) that contains information used to process the file. This structure includes a **file descriptor** (p. 541)—an index into an operating-system array called the **open file table** (p. 541). Each array element contains a **file control block** (FCB; p. 541) that the operating system uses to administer a particular file.
- The standard input, standard output and standard error are manipulated using the predefined file pointers **stdin**, **stdout** and **stderr**.
- Function **fgetc** (p. 541) **reads one character** from a file. It receives as an argument a FILE pointer for the file from which a character will be read.
- Function **fputc** (p. 541) **writes one character** to a file. It receives as arguments a character to be written and a FILE pointer for the file to which the character will be written.
- Functions fgets and fputs (p. 541) read a line from a file or write a line to a file, respectively.

Section 11.3 Creating a Sequential-Access File

- C imposes no structure on a file. You must provide a file structure to meet the requirements of a particular application.
- A C program administers each file with a separate FILE structure.
- Each open file must have a separately declared **FILE pointer** that's used to refer to the file.
- Function **fopen** (p. 543) takes as arguments a filename and a **file-open mode** (p. 543) and returns a pointer to the FILE structure for the file opened or NULL if the file could not be opened.
- The **file-open mode** "w" is used to open a file for writing. If the file does not exist, fopen creates it. If the file exists, the contents are discarded without warning.
- Function **feof** (p. 544) receives a pointer to a FILE and returns a nonzero (true) value when the end-of-file indicator has been set; otherwise, the function returns zero. Any attempt to read from a file for which feof returns true will fail.
- Function **fprintf** (p. 544) is equivalent to printf but also receives as an argument a file pointer for the file to which the data will be written.
- Function fclose (p. 544) receives a file pointer as an argument and closes the specified file.
- When a file is opened, the file control block (FCB) for the file is copied into memory. The FCB is used by the operating system to administer the file.
- To read an existing file, open it for reading ("r").
- To add records to the end of an existing file, open the file for appending ("a").
- To open a file for reading and writing, use an **update mode—"r+"**, "w+" or "a+". Mode "r+" opens a file for reading and writing. Mode "w+" creates a file for reading and writing, but an existing file's contents are discarded. Mode "a+" opens a file for reading and writing—all writing is done at the end of the file. If the file does not exist, it's created.
- Each file-open mode has a corresponding binary mode (b) for manipulating binary files.
- Exclusive write mode ensures that an existing file is not overwritten. If your program successfully opens a file in exclusive write mode and the underlying system supports exclusive file access, then only your program can access the file while it's open.

Section 11.4 Reading Data from a Sequential-Access File

- Function **fscanf** (p. 547) is equivalent to function scanf but receives as an argument a file pointer for the file from which the data is read.
- Function **rewind** repositions a program's **file position pointer** (p. 548) to the beginning of the file (i.e., byte 0) pointed to by its argument.
- The file position pointer is an integer value that specifies the byte location in the file at which the next read or write is to occur. This is sometimes referred to as the file offset (p. 548). The file position pointer is a member of the FILE structure associated with each file.
- The data in a sequential text file typically cannot be modified without the risk of destroying other data in the file.

Section 11.5 Random-Access Files

- Random-access files (p. 552) use fixed-length records that may be accessed directly without searching through other records.
- Every record in a random-access file normally has the same length, so the exact location of a record relative to the beginning of the file can be calculated as a function of the record key.
- Fixed-length records enable data to be inserted in a random-access file without destroying other data. Data stored previously can also be updated or deleted without rewriting the entire file.

Section 11.6 Creating a Random-Access File

- Function **fwrite** (p. 552) transfers a specified number of bytes beginning at a specified location in memory to a file. The data is written beginning at the file position pointer's location.
- Function **fread** (p. 552) transfers a specified number of bytes from the location in the file specified by the file position pointer to an area in memory beginning with a specified address.
- Functions fwrite and fread are capable of reading and writing arrays of data from and to files. The third argument of both fread and fwrite is the number of elements to process.
- File-processing programs normally write one struct at a time.

Section 11.7 Writing Data Randomly to a Random-Access File

- Function **fseek** (p. 555) repositions a file's file position pointer to a specific byte position. Its second argument indicates the number of bytes to seek, and its third argument indicates the location from which to seek. The third argument can be—SEEK_SET, SEEK_CUR or SEEK_END. **SEEK_SET** (p. 557) indicates that the seek starts at the beginning of the file; **SEEK_CUR** (p. 558) indicates that the seek starts at the current location in the file; and **SEEK_END** (p. 558) indicates that the seek is measured from the end of the file.
- Industrial-strength programs should determine whether functions such as fscanf, fseek and fwrite operate correctly by checking their return values.
- Function fscanf returns the number of fields successfully read or the value EOF if a problem occurs while reading data.
- Function fseek returns a nonzero value if the seek operation cannot be performed.
- Function fwrite returns the number of items it successfully output. If this number is less than the third argument in the function call, then a write error occurred.

Section 11.8 Reading Data from a Random-Access File

- Function fread reads a specified number of bytes from a file into memory.
- Function fread can read several fixed-size array elements by providing a pointer to the array in which the elements will be stored and by indicating the number of elements to be read.
- Function fread returns the number of items it successfully input. If this number is less than the third argument in the function call, then a read error occurred.

Self-Review Exercises

11.1	Fill-In the blanks in each of the following:
	a) Function closes a file.
	b) The function reads data from a file in a manner similar to how
	scanf reads from stdin.
	c) Function reads a character from a specified file.
	d) Function reads a line from a specified file.
	e) Function opens a file.
	f) Function is normally used when reading data from a file in random-
	access applications.
	g) Function repositions the file position pointer to a specific location
	in the file.

- 11.2 State which of the following are true and which are false. If false, explain why.
 - a) Function fscanf cannot be used to read data from the standard input.

- b) You must explicitly use fopen to open the standard input, standard output and standard error streams.
- c) A program must explicitly call function fclose to close a file.
- d) If the file position pointer points to a location in a sequential file other than the beginning of the file, the file must be closed and reopened to read from the beginning of the file.
- e) Function fprintf can write to the standard output.
- f) Data in sequential-access files can be updated without overwriting other data.
- g) It's not necessary to search through all the records in a random-access file to find a specific record.
- h) Records in random-access files are not of uniform length.
- i) Function fseek may seek only relative to the beginning of a file.
- 11.3 Write a single statement to accomplish each of the following. Assume that each of these statements applies to the same program.
 - a) Open the file "oldmast.dat" for reading and assign the returned file pointer to ofPtr.
 - b) Open the file "trans.dat" for reading and assign the returned file pointer to tfPtr.
 - c) Open the file "newmast.dat" for writing (and creation) and assign the returned file pointer to nfPtr.
 - d) Read a record from the file "oldmast.dat". The record consists of integer account, string name and floating-point currentBalance.
 - e) Read a record from the file "trans.dat". The record consists of the integer account and floating-point dollarAmount.
 - f) Write a record to the file "newmast.dat". The record consists of the integer account, string name and floating-point currentBalance.
- 11.4 Find the error in each of the following and explain how to correct it.
 - a) The file referred to by fPtr ("payables.dat") has not been opened.

```
printf(fPtr, "%d%s%d\n", account, company, amount);
```

- b) open("receive.dat", "r+");
- c) The following should read a record from "payables.dat". File pointer payPtr refers to this file, and file pointer recPtr refers to the file "receive.dat":

```
scanf(recPtr, "%d%s%d\n", &account, company, &amount);
```

d) The file "too1s.dat" should be opened to add data to the file without discarding the current data.

```
if ((tfPtr = fopen("tools.dat", "w")) != NULL)
```

e) The file "courses.dat" should be opened for appending without modifying the current contents of the file.

```
if ((cfPtr = fopen("courses.dat", "w+")) != NULL)
```

Answers to Self-Review Exercises

11.1 a) fclose. b) fscanf. c) fgetc. d) fgets. e) fopen. f) fread. g) fseek.

11.2 See the answers below:

- a) *False*. Function fscanf can be used to read from the standard input by including the pointer to the standard input stream, stdin, in the call to fscanf.
- b) *False*. These three streams are opened automatically by C when program execution begins.
- c) *False*. The files will be closed when program execution terminates, but all files should be explicitly closed with fclose.
- d) *False*. Function rewind can be used to reposition the file position pointer to the beginning of the file.
- e) True.
- f) *False*. In most cases, sequential file records are not of uniform length. Therefore, it's possible that updating a record will cause other data to be overwritten.
- g) True.
- h) False. Records in a random-access file are normally of uniform length.
- i) *False*. It's possible to seek from the beginning of the file, from the end of the file and from the current location in the file.

```
11.3 a)ofPtr = fopen("oldmast.dat", "r");
```

- b) tfPtr = fopen("trans.dat", "r");
- c) nfPtr = fopen("newmast.dat", "w");
- d) fscanf(ofPtr, "%d%s%f", &account, name, ¤tBalance);
- e) fscanf(tfPtr, "%d%f", &account, &dollarAmount);
- f) fprintf(nfPtr, "%d %s %.2f", account, name, currentBalance);

11.4 See the answers below:

- a) Error: "payables.dat" has not been opened before using fPtr. Correction: Use fopen to open "payables.dat" for writing, appending or updating.
- b) Error: Function open is not a Standard C function.
 - Correction: Use function fopen.
- c) Error: The function scanf should be fscanf. Function fscanf uses the incorrect file pointer to refer to file "payables.dat".
 - Correction: Use payPtr to refer to "payables.dat" and use fscanf.
- d) Error: The contents of the file are discarded because the file is opened for writing ("w").
 - Correction: To add data to the file, either open the file for updating ("r+") or open the file for appending ("a" or "a+").
- e) Error: File "courses.dat" is opened for updating in "w+" mode, which discards the current contents of the file.
 - Correction: Open the file in "a" or "a+" mode.

Exercises

- 11.5 Fill-In the blanks in each of the following:
 - a) Large amounts of data are stored on secondary storage devices as _____.
 - b) A(n) _____ is composed of several fields.

- c) To facilitate the retrieval of specific records from a file, one field in each record is chosen as a(n) ______.
 d) A group of related characters that conveys meaning is called a(n) ______.
 e) The file pointers for the three streams that are opened automatically when program execution begins are named ______, ____ and _____.
 f) Function _____ writes a character to a specified file.
 g) Function _____ writes a line to a specified file.
 h) Function _____ is generally used to write data to a random-access file.
- i) Function _____ repositions the file-position pointer to the beginning of the file.

11.6 (Creating Data for a File-Matching Program) Write a simple program to create some test data for checking out the program of Exercise 11.7. Use the following sample account data:

Master File: Account number	Name	Balance	Transaction File: Account number	Dollar amount
100	Alan Jones	348.17	100	27.14
300	Mary Smith	27.19	300	62.11
500	Sam Sharp	0.00	400	100.56
700	Suzy Green	-14.22	900	82.17

11.7 (File Matching) Exercise 11.3 asked you to write a series of single statements. These statements form the core of an important type of file-processing program, namely, a file-matching program. In commercial data processing, it's common to have several files in each system. In an accounts-receivable system, for example, there's generally a master file containing detailed information about each customer such as the customer's name, address, telephone number, outstanding balance, credit limit, discount terms, contract arrangements and possibly a condensed history of recent purchases and cash payments.

As transactions occur, such as sales and payments, they're entered into a file. At the end of each business period (i.e., a month for some companies, a week for others and a day in some cases), the file of transactions (called "trans.dat" in Exercise 11.3) is applied to the master file (called "oldmast.dat" in Exercise 11.3), to update each account's purchase and payment record. During an update, the master file is rewritten as a new file ("newmast.dat"), which is then used at the end of the next business period to begin the updating process again.

File-matching programs must deal with certain problems that do not exist in single-file programs. For example, a match does not always occur. A customer on the master file might not have made any purchases or cash payments in the current business period, and therefore no record for this customer will appear on the transaction file. Similarly, a customer who did make some purchases or cash payments might

have just moved to this community, and the company may not have had a chance to create a master record for this customer.

Use the statements in Exercise 11.3 as the basis for a complete file-matching accounts-receivable program. Use the account number on each file as the record key for matching purposes. Assume that each file is a sequential file with records stored in increasing account-number order.

When a match occurs (i.e., records with the same account number appear on both the master file and the transaction file), add the dollar amount on the transaction file to the current balance on the master file and write the "newmast.dat" record. (Assume that purchases are indicated by positive amounts on the transaction file and that payments are indicated by negative amounts.) When there's a master record for a particular account but no corresponding transaction record, merely write the master record to "newmast.dat". When there's a transaction record but no corresponding master record, print the message "Unmatched transaction record for account number ..." (fill in the account number from the transaction record).

- **11.8** (*Testing the File-Matching Exercises*) Run the program of Exercise 11.7 using the files of test data created in Exercise 11.6. Check the results carefully.
- 11.9 (File Matching with Multiple Transactions) It's possible (actually common) to have several transaction records with the same record key. This occurs because a particular customer might make several purchases and cash payments during a business period. Rewrite your accounts-receivable file-matching program of Exercise 11.7 to provide for the possibility of handling several transaction records with the same record key. Modify the test data of Exercise 11.6 to include the following additional transaction records:

Account number	Dollar amount
300	83.89
700	80.78
700	1.53

11.10(Write Statements to Accomplish a Task) Write statements that accomplish each of the following. Assume that the structure

```
struct person {
   char lastName[15];
   char firstName[15];
   char age[4];
};
```

has been defined and that the file is already open for writing.

- a) Initialize the file "nameage.dat" so that there are 100 records with lastName = "unassigned", firstname = "" and age = "0".
- b) Input 10 last names, first names and ages, and write them to the file.
- c) Update a record; if there's no information in the record, tell the user "No info".

574

- d) Delete a record that has information by reinitializing that particular record.
- **11.11** (Hardware Inventory) You're the owner of a hardware store and need to keep an inventory that can tell you what tools you have, how many you have and the cost of each one. Write a program that initializes the file "hardware.dat" to 100 empty records, lets you input the data concerning each tool, enables you to list all your tools, lets you delete a record for a tool that you no longer have and lets you update any information in the file. The tool identification number should be the record number. Use the following information to start your file:

Record #	Tool name	Quantity	Cost
3	Electric sander	7	57.98
17	Hammer	76	11.99
24	Jig saw	21	11.00
39	Lawn mower	3	79.50
56	Power saw	18	99.99
68	Screwdriver	106	6.99
77	Sledge hammer	11	21.50
83	Wrench	34	7.50

11.12(*Telephone-Number Word Generator*) Standard telephone keypads contain the digits 0–9. The numbers 2–9 each have three letters associated with them, as is indicated by the following table:

Digit	Letter	Digit	Letter
2	АВС	6	MNO
3	DEF	7	PRS
4	GHI	8	TUV
5	JKL	9	WXY

Many people find it difficult to memorize phone numbers, so they use the correspondence between digits and letters to develop seven-letter words that correspond to their phone numbers. For example, a person whose telephone number is 686-2377 might use the correspondence indicated in the above table to develop the seven-letter word "NUMBERS".

Businesses frequently attempt to get telephone numbers that are easy for their clients to remember. If a business can advertise a simple word for its customers to dial, then, no doubt, the business will receive a few more calls.

Each seven-letter word corresponds to exactly one seven-digit telephone number. The restaurant wishing to increase its take-home business could surely do so with the number 825-3688 (i.e., "TAKEOUT").

Each seven-digit phone number corresponds to many separate seven-letter words. Unfortunately, most of these represent unrecognizable juxtapositions of let-

ters. It's possible, however, that the owner of a barbershop would be pleased to know that the shop's telephone number, 424-7288, corresponds to "HAIRCUT". The owner of a liquor store would, no doubt, be delighted to find that the store's telephone number, 233-7226, corresponds to "BEERCAN". A veterinarian with the phone number 738-2273 would be pleased to know that the number corresponds to the letters "PETCARE".

Write a C program that, given a seven-digit number, writes to a file every possible seven-letter word corresponding to that number. There are 2187 (3 to the seventh power) such words. Avoid phone numbers with the digits 0 and 1.

- **11.13**(*Project: Telephone-Number Word Generator Modification*) If you have a computerized dictionary available, modify the program you wrote in Exercise 11.12 to look up the words in the dictionary. Some seven-letter combinations created by this program consist of two or more words (e.g., the phone number 843-2677 produces "THEBOSS").
- **11.14**(*Using File-Processing Functions with Standard Input/Output Streams*) Modify the example of Fig. 8.8 to use functions fgetc and fputs rather than getchar and puts. The program should give the user the option to read from the standard input and write to the standard output or to read from a specified file and write to a specified file. If the user chooses the second option, have the user enter the filenames for the input and output files.
- 11.15 (Outputting Type Sizes to a File) Write a program that uses the sizeof operator to determine the sizes in bytes of the various data types on your computer system. Write the results to the file "datasize.dat" so you may print the results later. The format for the results in the file should be as follows (the type sizes on your computer might be different from those shown in the sample output):

Data type	Ci 70
Data type	Size
char	1
unsigned char	1
short int	2
unsigned short int	2
int	4
unsigned int	4
long int	4
unsigned long int	4
float	4
double	8
long double	16
_	

11.16(Simpletron with File Processing) In Exercise 7.29, you wrote a software simulation of a computer that used a special machine language called Simpletron Machine Language (SML). In the simulation, each time you wanted to run an SML program, you entered the program into the simulator from the keyboard. If you made a mistake while typing the SML program, the simulator was restarted, and the SML code was reentered. It would be nice to be able to read the SML program from a file rather than

type it each time. This would reduce time and mistakes in preparing to run SML programs.

- a) Modify the simulator you wrote in Exercise 7.29 to read SML programs from a file specified by the user at the keyboard.
- b) After the Simpletron executes, it outputs the contents of its registers and memory on the screen. It would be nice to capture the output in a file, so modify the simulator to write its output to a file in addition to displaying it on the screen.
- **11.17** (Modified Transaction-Processing System) Modify the program of Section 11.9 to include an option that displays the list of accounts on the screen. Consider modifying function textFile to use either the standard output or a text file based on an additional function parameter that specifies where the output should be written.
- **11.18**(*Project: Phishing Scanner*) Phishing is a form of identity theft in which, in an e-mail, a sender posing as a trustworthy source attempts to acquire private information, such as your user names, passwords, credit-card numbers and Social Security number. Phishing e-mails claiming to be from popular banks, credit-card companies, auction sites, social networks and online payment services may look quite legitimate. These fraudulent messages often provide links to spoofed (fake) websites where you're asked to enter sensitive information.

Visit https://snopes.com and other websites to find lists of the top phishing scams. Also, check out the Anti-Phishing Working Group (https://apwg.org/), and the FBI's Cyber Investigations website (https://www.fbi.gov/investigate/cyber), where you'll find information about the latest scams and how to protect yourself.

Create a list of 30 words, phrases and company names commonly found in phishing messages. Assign a point value to each based on your estimate of its likeliness to be in a phishing message (e.g., one point if it's somewhat likely, two points if moderately likely, or three points if highly likely). Write a program that scans a file of text for these terms and phrases. For each occurrence of a keyword or phrase within the text file, add the assigned point value to the total points for that word or phrase. For each keyword or phrase found, output one line with the word or phrase, the number of occurrences and the point total. Then show the point total for the entire message. Does your program assign a high point total to some actual phishing e-mails you've received? Does it assign a high point total to some legitimate e-mails you've received?

Al Case Study: Intro to NLP—Who Wrote Shakespeare's Works?

11.19 (Intro to Natural Language Processing and Similarity Detection) Every day, we use natural language in various forms of communication, including:

- You read your text messages and check the latest news clips.
- You speak to family, friends and colleagues and listen to their responses.

- You have a hearing-impaired friend with whom you communicate via sign language and who enjoys close-captioned video programs.
- You have a blind colleague who reads braille, listens to audiobooks and listens to a screen reader speak about what's on the computer screen.
- You read e-mails, distinguishing junk from important communications.
- You receive a client e-mail in Spanish and run it through a free translation program, then respond in English, knowing that your client can easily translate your e-mail back to Spanish.
- You drive, observing road signs like "Stop," "Speed Limit 35" and "Road Under Construction."
- You give your car verbal commands, like "call home" or "play classical music," or ask questions like, "Where's the nearest gas station?"
- You teach a child how to speak and read.
- You learn a foreign language.

Natural Language Processing (NLP) helps computers understand, analyze and process human text and speech. Natural language processing is performed on text collections composed of Tweets, Facebook posts, conversations, movie reviews, Shakespeare's plays, historical documents, news items, meeting logs, and so much more. A text collection is known as a corpus, the plural of which is corpora.

Some key NLP applications include:

- Natural language understanding—understanding text content or spoken language.
- Sentiment analysis—determining whether text has positive, neutral or negative sentiment. For example, companies analyze the sentiment of tweets about their products.
- Readability assessment—determining how readable text is, based on the vocabulary used, word lengths, sentence lengths, sentence structure, topics covered and more. While writing this book, we used the paid NLP tool Grammarly² to help us tune the writing to ensure the text's readability for a wide audience.
- Intelligent virtual assistants—software that helps you perform everyday tasks. Popular intelligent virtual assistants include Amazon Alexa, Apple Siri, Microsoft Cortana and Google Assistant.
- Text summarization—summarizing the key points of a large text. This can save valuable time for busy people.
- Speech recognition—converting speech to text.
- Speech synthesis—converting text to speech.

^{2.} Grammarly also has a free version (https://www.grammarly.com).

- Language identification—receiving a text when you don't know its language in advance then automatically determining the language.
- Interlanguage translation—converting text to other spoken languages.
- Named-entity recognition—locating and categorizing items like dates, times, quantities, places, people, things, organizations and more.
- Chatbots—AI-based software that humans interact with via natural language. One popular chatbot application is automated customer support.
- Similarity detection—examining documents to determine how alike they are. Basic similarity metrics include average sentence length, frequency distribution of sentence lengths, average word length, frequency distribution of word lengths, frequency distribution of word usage, and more.

Many lower-level NLP tasks support the applications above as they perform their tasks, including:

- Tokenization—splitting text into tokens, which are meaningful units, such as words and numbers.
- Parts-of-speech (POS) tagging—identifying each word's part of speech, such as noun, verb, adjective, etc.
- Noun phrase extraction—locating groups of words representing nouns, such as "red brick factory."³
- Spell checking and spelling correction.
- Stemming—reducing words to their stems by removing prefixes or suffixes. For example, the stem of "varieties" is "varieti."
- Lemmatization—like stemming, but produces real words based on the original words' context. For example, the lemmatized form of "varieties" is "variety."
- Word frequency counting—determining how often each word appears in a corpus.
- **Stop-word elimination**—removing common words, such as *a*, *an*, *the*, *I*, *we*, *you* and more to analyze the important words in a corpus.
- n-grams—producing sets of consecutive words in a corpus for use in identifying words that frequently appear adjacent to one another. n-grams are commonly used for predictive text input, such as when your smartphone suggests possible next words as you type a text message.

This case study exercise serves two purposes:

• First, it introduces the crucial AI subtopic of natural language processing, which will play a key role in the future of anyone learning programming today.

^{3.} The phrase "red brick factory" illustrates why natural language is such a difficult subject. Is a "red brick factory" a factory that makes red bricks? Is it a red factory that makes bricks of any color? Is it a factory built of red bricks that makes products of any type? In today's music world, it could even be the name of a rock band or the name of a game on your smartphone.

• Second, it introduces the NLP subtopic of similarity detection, which you'll perform using straightforward array-, string- and file-processing techniques.

Project Gutenberg

A great source of text for analysis is the massive collection of free e-books at Project Gutenberg:

```
https://www.gutenberg.org
```

The site contains over 60,000 e-books in various formats, including plain-text files. These are out of copyright in the United States. For information about Project Gutenberg's Terms of Use and copyright in other countries, see:

```
https://www.gutenberg.org/policy/terms_of_use.html
```

For this case-study exercise, you'll use the plain-text e-book files for William Shakespeare's *Romeo and Juliet*:

```
https://www.gutenberg.org/ebooks/1513
```

and Christopher Marlowe's Edward the Second:

```
https://www.gutenberg.org/ebooks/20288
```

Each of these is available free for download at Project Gutenberg.

Downloading E-Books from Project Gutenberg

Project Gutenberg does not allow programmatic access to its e-books. You must download the books to your own system before analyzing them.⁴ To download *Romeo and Juliet* as a plain-text file, right-click the Plain Text UTF-8 link on the play's web page, then select

- Save Link As... (Chrome/Firefox/Microsoft Edge),
- Download Linked File As... (Safari), or

to save the play to the folder in which you'll place your solution to this exercise. Save the files with the names RomeoAndJuliet.txt and EdwardTheSecond.txt.

Who Wrote Shakespeare's Works?

Some people believe that William Shakespeare's works might have been penned by Christopher Marlowe, Sir Francis Bacon or others. You can learn more about this controversy at

```
https://en.wikipedia.org/wiki/Shakespeare_authorship_question
```

With some simple similarity-detection techniques, you can begin to compare Shake-speare's works with those of other authors. In this case-study exercise, your ultimate goal is to perform similarity detection between *Romeo and Juliet* and Christopher Marlowe's *Edward the Second* to determine whether Christopher Marlowe might have

^{4. &}quot;Information About Robot Access to our Pages." Accessed January 1, 2021. https://www.gutenberg.org/policy/robot_access.html.

authored Shakespeare's works. If you really get into this issue, you can explore more sophisticated similarity-detection techniques.

Analyzing Romeo and Juliet to Prepare for Simple Similarity Detection

You'll now perform some simple statistical analysis as a basis for determining document similarity. You'll begin by focusing on Shakespeare's *Romeo and Juliet*. Later, you'll perform the same tasks on *Edward the Second*, then compare your analyses' results. As a control, you also might want to analyze a play from a third author who is not involved in this controversy. You'll track the following items as you read and process *Romeo and Juliet*, then use them to display various statistics:

- The total number of sentences.
- The total number of words.
- The total number of characters.
- The number of sentences of each length.
- The number of words of each length.
- The unique words' frequencies.

Cleaning Romeo and Juliet Before Analyzing It

Data does not always come ready for analysis. It could, for example, be in the wrong format. Data scientists spend a large portion of their time preparing data before performing analyses. Preparing data for analysis is called data munging or data wrangling.

Each e-book you download from Project Gutenberg contains information and legal paragraphs that you will not want to include in your analyses. You should open *Romeo and Juliet* in a text editor and "clean" it by removing the Project Gutenberg text. In particular, remove everything from the beginning of the document up to and including the title "THE TRAGEDY OF ROMEO AND JULIET," then remove everything from the following text through the end of the file:

End of the Project Gutenberg EBook of Romeo and Juliet, by William Shakespeare

You should do some additional text cleaning with a text editor before running your analytics on the play:

- Each character's name is mentioned each time that character speaks—this is standard in plays. You don't need the characters' names for the particular analytics you're going to run in this case study. In fact, they'll "get in the way." For more sophisticated similarity detection, you might want to keep them.
- Plays also include many staging directions indicating when characters should enter and leave the stage, duel one another, fall down and die when poisoned, and the like. These directions also should be removed.

You could write a program to handle these cleaning chores. Be careful, though—scrutinizing the manuscript may reveal many special cases that your code would need to handle. Programming for them could be time-consuming and error-prone.

Arrays You'll Need to Perform Your Analysis

You are now ready to analyze *Romeo and Juliet* to create the statistics you'll use for simple similarity detection. Use three arrays to record various counts that characterize the text of the play.

- The **sentenceLengths** array will keep counts of how many sentences consist of one word, two words, three words, etc.
- The wordLengths array will keep counts of how many words consist of one character, two characters, three characters, etc.
- The wordFrequencies array contains a struct for each distinct word in the play. The struct's members are the word and a count of how many times that word appears in the play. The word should be stored as a string in a fixed-length char array, which must be large enough to store the longest word in the play and its terminating null character.

Analyzing a Sentence

Consider the sentence:

"O Romeo, Romeo, wherefore art thou Romeo."

- The sentence has seven words, so your program would add 1 to sentenceLengths[7].
- The first word ("O") has one letter, so your program would add 1 to word-Lengths[1].
- The second word ("Romeo") has five letters, so your program would add 1 to wordLengths [5], and so on.

To perform word-frequency counting, convert the words to lowercase letters, so that all occurrences of the same word will compare as equal. As you process each word, search the wordFrequencies array to determine whether the word already is in the array. If so, add one to that word's count. Otherwise, place the word in the next empty wordFrequencies array element and set its count to 1.

Implementing Your Analysis Code

Use the array-, string- and file-processing techniques you've learned to read the contents of *Romeo and Juliet* and perform the following tasks:

- Every sentence ends with a sentence terminator—a period (.), a question mark (?) or an exclamation point (!). Define a processSentence function that reads words until it encounters a sentence terminator. This function updates the sentence, word and character counters as you process each word. When you hit the end of a sentence, increment the appropriate counter in the sentenceLengths array and reset the word counter to zero.
- For every word, processSentence should call the **processWord function** to increment the appropriate counter in the wordLengths array, and either incre-

ment the word's counter in the wordFrequencies array or add the word to the wordFrequencies array with a count of 1.

Remember to keep track of the total number of sentences, words and characters.

Analysis Report

Next, display the following statistics for *Romeo and Juliet*:

- The total number of sentences.
- The total number of words.
- The total number of characters.
- The mean (average) sentence length.
- The mean word length.
- The median sentence length.
- The median word length.
- A table of sentence lengths and their percentages among all sentence lengths.
- A table of word lengths and their percentages among all word lengths.
- A frequency-distribution table containing the play's unique words, their frequencies and their percentages among all words in the play—display these in descending order by frequency.

Your program also should output these statistics to a file to make it easier to study the results you produce and compare them between plays.

Analyzing Christopher Marlowe's Play Edward the Second

Now that you've analyzed *Romeo and Juliet*, use a text editor to clean Christopher Marlowe's play *Edward the Second*. As part of any data-science study, it's important to get to know your data. The conventions used in *Edward the Second* for specifying who's speaking and the play's staging directions are different from those you saw in *Romeo and Juliet*. So, be careful to observe these differences when cleaning *Edward the Second*. After you clean the text, run your analytics program on *Edward the Second*. Compare the analytics with those you produced for *Romeo and Juliet*. Comment on the similarities you find between these plays.

AI/Data-Science Case Study—Machine Learning with GNU Scientific Library

11.20 (Machine Learning with Simple Linear Regression) Machine learning is one of the most exciting and promising subfields of artificial intelligence. Our goal here is to give you a friendly, hands-on introduction to one of the simpler machine-learning techniques.

Prediction

Machine learning is typically used to make predictions, based on existing data—and often lots of it. Wouldn't it be fantastic if you could improve weather forecasting to

save lives, minimize injuries and property damage? What if we could improve cancer diagnoses and treatment regimens to save lives, or improve business forecasts to maximize profits and secure people's jobs? What about detecting fraudulent credit-card purchases and insurance claims? How about predicting customer "churn," what prices houses are likely to sell for, ticket sales of new movies, and more generally, anticipated revenue of new products and services? How about predicting the best strategies for coaches and players to use to win more games and championships? All of these kinds of predictions are happening today with machine learning.

GNU Scientific Library and gnuplot

In this case study, you'll examine a completely coded program that demonstrates the machine-learning technique called simple linear regression, performed with a function from the open-source GNU Scientific Library:

https://www.gnu.org/software/gsl/

This library defines many commonly used algorithms from engineering, science and mathematics. The program you'll study then passes commands to the 2D and 3D plotting application **gnuplot** to create several plot images. As you'll see, gnuplot uses its own plotting language different from C, so in our code, we provide extensive comments that explain the gnuplot commands.

Descriptive Statistics

In data science, you'll often use statistics to describe and summarize your data. Some basic descriptive statistics are:

- minimum—the smallest value in a collection of values.
- maximum—the largest value in a collection of values.
- range—the difference between the maximum and minimum values.
- count—the number of values in a collection.
- **sum**—the total of the values in a collection.

Measures of dispersion (also called measures of variability), such as *range*, determine how spread out values are. Other measures of dispersion include *variance* and *standard deviation*. 5,6,7

Additional descriptive statistics include mean, median and mode, which we discussed in Section 6.9. These are measures of central tendency—each is a way of producing a single value that represents a "central" value in a set of values, i.e., one which is in some sense typical of the others.

^{5. &}quot;Understanding Descriptive Statistics." Accessed January 1, 2021. https://towardsdata-science.com/understanding-descriptive-statistics-c9c2b0641291.

^{6. &}quot;Standard deviation." Accessed January 1, 2021. https://en.wikipedia.org/wiki/Standard_deviation.

^{7. &}quot;Variance." Accessed January 1, 2021. https://en.wikipedia.org/wiki/Variance.

Anscombe's Quartet

An important step in data analytics is "getting to know your data." The *basic descriptive statistics* above certainly help you know more about your data. One caution, though, is that dramatically different datasets actually can have identical or nearly identical descriptive statistics. For an example of this phenomenon, we'll consider *Anscombe's Quartet*:

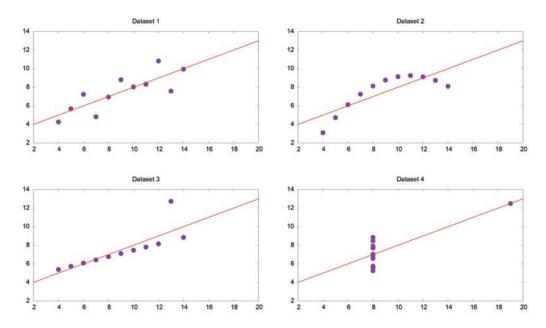
https://en.wikipedia.org/wiki/Anscombe%27s_quartet
Anscombe's Quartet consists of the following four sets of *x-y* coordinate pairs with 11

data samples each:

хI	уI	x2	y2	х3	у3	x4	y4
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

Interestingly, these datasets have *nearly identical* descriptive statistics. For instance, in all four datasets, the *x*- and *y*-coordinates' mean values are 9 and 7.5, respectively.

The following diagrams—which our fully coded case study example creates—plot the x1-y1, x2-y2, x3-y3 and x4-y4 data, respectively:



We'll discuss the lines (known as **regression lines**) shortly. As you can see in the **visualizations**—but not necessarily by simply looking at the data in the preceding table—these Anscombe's Quartet datasets are *significantly different*. Yet, like their descriptive statistics, their regression lines appear to be identical. This shows that you cannot just draw conclusions from descriptive statistics and regressions. You must use additional tools, like the visualizations above, to **get to know your data**.

Simple Linear Regression

Given a collection of points (x-y coordinate pairs) representing an independent variable (x) and a dependent variable (y), simple linear regression describes the linear relationship between the dependent and independent variables with a straight line, known as the regression line. The lines in the preceding diagrams are the regression lines for each of the four datasets in Anscombe's Quartet.

Consider the linear relationship between Celsius and Fahrenheit temperatures. Given a Celsius temperature, we can calculate the corresponding Fahrenheit temperature using the following formula:

```
fahrenheit = 9 / 5 * celsius + 32
```

In this formula, celsius is the *independent variable*, and fahrenheit is the *dependent variable*. Each fahrenheit temperature *depends on* the celsius temperature used in the calculation. If we were to plot the Fahrenheit temperature for each Celsius temperature, all of the points would appear along the same straight line, revealing a *linear relationship* between the two temperature scales.

Components of the Simple-Linear-Regression Equation

The points along any straight line (in two dimensions) like the regression lines shown in the preceding diagrams can be calculated with the equation:

$$y = mx + b$$

where

- *m* is the line's **slope**,
- b is the line's intercept with the y-axis (at x = 0), or simply the y-intercept,
- *x* is the independent variable, and
- *y* is the dependent variable.

In the formula for converting Celsius temperatures to Fahrenheit temperatures:

- m is 9 / 5,
- b is 32,
- x is celsius—the independent Celsius temperature, and
- y is fahrenheit—dependent Fahrenheit temperature produced by the calculation.

In simple linear regression, *y* is the *predicted value* for a given *x*. Of course, a line has an infinite number of points. If you can determine with simple linear regression the

equation for a straight line from a modest finite number of sample points, you then have the means to make an infinite number of predictions, even for independent variable values you've never seen before.

How Simple Linear Regression Works

Simple linear regression is a machine-learning technique that determines the slope (m) and y intercept (b) of a straight line that "best fits" your data. The simple linear regression algorithm iteratively adjusts the slope and intercept and, for each adjustment, calculates the square of each point's distance from the line. The "best fit" occurs when the slope and intercept values minimize the sum of those squared distances. This is known as an ordinary least squares calculation.⁸

Performing Simple Linear Regression with the GNU Scientific Library

The GNU Scientific Library's gs1_fit_linear function encapsulates simple linear regression's calculations, giving you as results the slope and y-intercept for the straight line that best fits the data. After calling gs1_fit_linear, you can plug into the y = mx + b equation the slope (m) and intercept (b), then predict dependent y values, based on independent x values. We also use these values with **gnuplot** to display the regression line for the data along with the data points.

Comma-Separated Values (CSV) Files

We provided the Anscombe's Quartet data for you in the file anscombe.csv. This file and this case-study exercise's source code are located in the AnscombesQuartet subfolder of this chapter's examples folder. The .csv filename extension indicates that the file is in CSV (comma-separated values) format—a particularly popular file format for distributing datasets. CSV files are simply text files in which each line is one record of information with its items separated by commas. The following are the first two rows of anscombe.csv:

```
x1,y1,x2,y2,x3,y3,x4,y4
10,8.04,10,9.14,10,7.46,8,6.58
```

A CSV file's first row typically contains column names for the data in subsequent rows. In anscombe.csv, the remaining rows are the numeric values for the columns. In our code, the function readAnscombesQuartetData loads the data into arrays.

Installing the GNU Scientific Library on macOS

On macOS, you can install the GNU Scientific Library using the Homebrew package manager⁹ as follows:

brew install gsl

Installing the GNU Scientific Library on Windows

In Visual Studio, you add the GNU Scientific Library to each project in which you wish to use it. With your project open in Visual Studio, perform the following steps:

^{8.} https://en.wikipedia.org/wiki/Ordinary_least_squares.

^{9.} If the brew command is not found, visit https://brew.sh/ for install instructions.

- 1. Select Tools > NuGet Package Manager > Manage NuGet Packages for Solution....
- 2. In the Browse tab, search for "gsl-msvc-", then select gsl-msvc-x64.
- **3.** In the right side of the NuGet package manager, click the checkbox next to your project's name, then click **Install** to add the library to your project.

Installing gnuplot on macOS

Install the gnuplot using the Homebrew package manager as follows:

brew install gnuplot

Installing gnuplot on Windows

Download and run the gnuplot Windows installer (gp541-win64-mingw.exe) from:

https://sourceforge.net/projects/gnuplot/files/gnuplot/5.4.1/

Click Next > until you reach the Select Additional Tasks step, then:

- Under Select gnuplot's default terminal, select the windows radio button.
- Scroll to the bottom of the settings and check the Add application directory to your PATH environment variable checkbox.

Click Next >, then Install. Once the installation completes, reboot your computer.

Compiling and Running the Program on macOS

On macOS, compile anscome_macos.c by performing the following steps:

- **1.** Open a Terminal window.
- 2. Use the cd command to change to AnscombesQuartet subfolder of this chapter's examples folder.
- **3.** Compile the program with the following command:

```
clang -std=c18 anscombe_macos.c -lgsl -o anscombe_macos
```

4. Run the program:

./anscombe_macos

The program will create four PNG image files in the same folder as anscome_macos.c on macOS. You can open these image files to view the four plots.

Compiling and Running the Program on Windows

In the Visual Studio solution where you added the GNU Scientific library:

- Add to your project the file anscome_windows.c from the AnscombesQuartet subfolder of this chapter's examples folder.
- Modify line 72 to specify the location of anscombe.csv on your system
- Build and run your project.

When you run the program it will create four PNG image files in your project's folder. Use File Explorer to navigate to that folder, then open the image files to view the four plots.

Extensively Commented Code

Next, study the code to learn how to use the gsl_fit_linear function of the GNU Scientific Library to perform simple linear regression, and how to send gnuplot commands from a C program to the gnuplot application. Consider tweaking the gnuplot commands to see how your changes affect the plots our program produces. For example, you can change the plot's pointtype, linewidth and linecolor values.

AI/Data-Science Case Study: Time Series and Simple Linear Regression

Now that you've carefully studied the code for Anscombe's Quartet, you can adapt the program to other simple-linear-regression problems. Simple linear regression is commonly used to analyze time series—sequences of values (called observations) associated with points in time. Some examples are daily closing stock prices, hourly temperature readings, the changing positions of a plane in flight, annual crop yields and quarterly company profits. Perhaps the ultimate time series is the stream of time-stamped tweets coming from Twitter users worldwide.

Time Series

For this exercise, you'll use simple linear regression to analyze a time series containing New York City's average January temperatures *ordered* by year for the years 1895–2020. This is a **univariate time series**—it contains *one* observation per time. A **multivariate time series** has *two or more* observations per time, such as hourly temperature, humidity and barometric-pressure readings in a weather application. Your goal in this exercise is to determine whether the regression line has:

- a negative slope, indicating a declining average temperature trend over that time,
- a zero slope, indicating a stable average temperature trend over that time, or
- a **positive slope**, indicating an **increasing average temperature trend** over that time.

Getting Weather Data from NOAA

The National Oceanic and Atmospheric Administration (NOAA)

http://www.noaa.gov

provides extensive public historical weather data, including time series for average temperatures in specific cities over various time intervals.

We obtained the New York City January average temperatures for 1895–2020 (the maximum date range available at the time of this writing) from NOAA's "Climate at a Glance" time series at:

https://www.ncdc.noaa.gov/cag/city/time-series

You can select weather data for the entire U.S., regions within the U.S., states, cities and more. After selecting the data you need and the time frame to analyze, click Plot to display a diagram and view a table of the selected data. At the top of that table are icons you can click to download the data in several formats, including CSV.

For your convenience, we provided the file nyc_ave_january_temps.csv containing the data you'll use in this exercise. The file is located in the nycdata subfolder of this chapter's examples folder. We also "cleaned" the data, so the file contains the following two columns per observation:

- Date—A value of the form YYYY (such as 2020). The downloaded data is in the form YYYYMM (such as 202001), where 01 represents January. We removed 01 from each data item in this column, leaving only the year.
- Temperature—A floating-point Fahrenheit temperature. We renamed this column from Value in the downloaded data.

We deleted a third column called Anomaly that's not required for this exercise.

Performing the Regression

Modify the Anscombe's Quartet code to perform simple linear regression using the New York City average January temperatures data and to plot the data with a regression line. What trend do you see over the last 126 years?

Web Services and the Cloud Case Study—libcurl and Open-WeatherMap

11.21 (Getting a City's Weather Report with OpenWeatherMap) This is another of our challenge case-study exercises. Section 1.11 introduced the Internet, the web, the cloud, web services and mashups. In this case-study exercise, you'll dive into the world of web services using the open-source libcurl¹⁰ and cJSON¹¹ libraries to invoke a web service and process the results it returns. You'll study a fully coded, heavily commented program that interacts with an OpenWeatherMap free web service from

https://openweathermap.org/

to get the current weather report for a city of your choosing.

We'll then challenge you to create your first mashup using what you've learned from this **fully coded example**. If you're entrepreneurial, you can quickly prototype powerful new applications by weaving existing web services into mashups. Even if you're not going to do the related challenge project, just mastering this case study's code will open up the vast world of web services to you.

Web Services

The machine on which a web service resides is referred to as a web-service host. A client application (in our case, a C program) sends a request over a network to the web-service host, which processes the request and returns a response over the network to the client. This kind of distributed computing benefits systems in various ways. For example, an application without direct access to data on another system might be

^{10. &}quot;libcurl — the multiprotocol file transfer library." Accessed January 4, 2021. https://curl.se/libcurl/.

^{11. &}quot;cJSON." Accessed January 4, 2021. https://github.com/DaveGamble/cJSON.

able to retrieve the data via a web service. Similarly, an application lacking the processing power to perform specific computations could use a web service to take advantage of another system's superior resources.

Representational State Transfer (REST)

Most web services today use an architectural style known as Representational State Transfer (REST) and are often called RESTful web services. In a RESTful web service, each function you can call is identified by a unique URL. URLs (Uniform Resource Locators) identify the locations on the Internet of resources, such as web sites and web services. When a web server receives a request to a RESTful web service, it immediately knows what function to call on that server. RESTful web services can be called from programs, as you'll do here, or directly from a web browser's address bar by entering the appropriate URL.

OpenWeatherMap

OpenWeatherMap provides a free tier to many of its weather web services. Before you can use them, you must sign up for a free account at https://openweathermap.org/. They will send you an email to verify your account. Once you do, they'll reply with an email that contains your free API key. You also can find this under the API Keys tab in your account when you log into the site.

You can view the variety of OpenWeatherMap APIs and their documentation at https://openweathermap.org/api

Some are free and some are available only to subscribers. With a free API key, you can access:

- the Current Weather Data for a specified location, which we use in this case study,
- the One Call API, which returns a combination of current and future weather data for a specified location, and
- the 5 Day / 3 Hour Forecast for a specified location.

JavaScript Object Notation (JSON)

Many cloud-based services like OpenWeatherMap communicate with your applications via JSON objects. JSON (JavaScript Object Notation) is a text-based, human-and-computer-readable, data-interchange format used to represent data as collections of name/value pairs. JSON has become the preferred data format for transmitting data over the Internet between applications. This is especially true for invoking cloud-based web services.

Each JSON object contains a comma-separated list of *property names* and *values* in curly braces. For example, the following key-value pairs might represent a client record:

```
{"account": 100, "name": "Jones", "balance": 24.98}
```

JSON also supports arrays, which are comma-separated values in square brackets. For example, the following is an acceptable JSON array of numbers:

```
[100, 200, 300]
```

Values in JSON objects and arrays can be:

- strings in double quotes (like "Jones"),
- numbers (like 100 or 24.98),
- JSON Boolean values (represented as true or false),
- null (to represent no value, like NULL in C),
- arrays (like [100, 200, 300]), and
- other JSON objects.

Figure 11.8 contains a sample JSON response from OpenWeatherMap's Current Weather Data web service, which we formatted for readability. Even though you may never have seen JSON-encoded data, you should find this to be well organized, readable and pretty understandable.

```
{
 "coord": {
 2
        "lon": -71.06,
 3
        "lat": 42.36
 4
 5
 6
      "weather": [
 7
           "id": 803,
 8
           "main": "Clouds",
 9
           "description": "broken clouds",
10
           "icon": "04n"
П
        }
12
13
      ],
      "base": "stations",
14
      "main": {
15
        "temp": 0.03,
16
        "feels_like": -4.96,
17
        "temp_min": -1.11,
18
19
        "temp_max": 1.11,
20
        "pressure": 1014,
21
        "humidity": 93
22
      "visibility": 10000,
23
      "wind": {
24
25
        "speed": 4.1,
        "deg": 360
26
27
       "clouds": {
28
29
        "all": 75
30
      "dt": 1609815037,
31
```

Fig. 11.8 | Sample OpenWeatherMap response for Boston, MA, USA. (Part 1 of 2.)

```
"sys": {
32
        "type": 1,
33
        "id": 3486,
34
        "country": US",
35
        "sunrise": 1609762409,
36
        "sunset": 1609795488
37
38
      "timezone": -18000,
39
      "id": 4930956,
40
      "name": "Boston",
41
      "cod": 200
42
    }
43
```

Fig. 11.8 | Sample OpenWeatherMap response for Boston, MA, USA. (Part 2 of 2.)

Open-Source libcurl Library

To obtain the JSON response in Fig. 11.8, our application uses functions from the open-source libcurl library:

```
https://curl.se/libcurl/
```

The library supports many Internet and web protocols for transmitting data between applications and can be used to invoke web services and receive their responses. You can find the documentation for libcurl's C functions at:

```
https://curl.se/libcurl/c/
```

In our fully coded example, weather.c, located in the weather folder of this chapter's examples folder, we extensively comment the libcurl functions you need to invoke a web service and save its response to a file.

To install libcurl on macOS or Linux:

• For macOS, you can install the libcurl library using the **Homebrew package** manager¹² as follows:

```
brew install libcurl4
```

• For Ubuntu Linux, execute the command

```
sudo apt install libcurl4-openssl-dev
```

In Visual Studio, you add libcurl to each project in which you wish to use it. With your project open in Visual Studio, perform the following steps:

- 1. Select Tools > NuGet Package Manager > Manage NuGet Packages for Solution....
- 2. In the Browse tab, search for "curl", then select "curl by curl contributors."
- **3.** In the right side of the **NuGet Package Manager**, click the checkbox next to your project's name, then click **Install** to add the library to your project.

Open Source cJSON Library

The libcurl part of our application writes the OpenWeatherMap JSON response to a file. To display the weather report, our app reads the file's contents into a string,

^{12.} If the brew command is not found, visit https://brew.sh/ for install instructions.

then uses the open-source cJSON library to extract items from the JSON. You can download cJSON from:

```
https://github.com/DaveGamble/cJSON
```

There is no installation procedure for this library. You simply include the library's cJSON.h and cJSON.c files in your project.

cJSON's functions enable you to access items in the JSON response so we can display a weather report like the following:

Boston Weather
Temperature: 0.0 C
Feels like: -5.0 C
Pressure: 1014 hPa
Humidity: 93%
Conditions: broken clouds

In weather.c, we extensively commented the cJSON functions you need to extract the data above for the city you specify when you run the application (discussed below).

Compiling and Running the Program on macOS and Linux

On macOS and Linux, compile the weather app by performing the following steps:

- 1. Open a Terminal window.
- 2. Use the cd command to change to weather subfolder of this chapter's examples folder.
- **3.** Compile the program with one of the following commands—clang on macOS or gcc on Linux:

```
clang -std=c18 -Wall weather.c cJSON.c -lcurl -o weather
gcc -std=c18 -Wall weather.c cJSON.c -lcurl -o weather
```

This application receives two command-line arguments. Though we do not discuss the details of command-line arguments until Section 15.3, this completely coded simulation provides the statements you need to receive the command-line arguments. The first is the city for which you'd like to get the current weather, such as

```
Boston, MA, USA
```

If the city's name contains a space, enclose the location in quotes:

```
"Los Angeles, CA, USA"
```

The second command-line argument is your OpenWeatherMap API key. The following command would get the current weather data for Boston, MA, USA:

```
./weather Boston,MA,USA API\_KEY
```

Be sure to replace API_KEY with the OpenWeatherMap API key you received when you signed up for your free account.

Compile the program and run it several times. Next, study this application's code (including the extensive comments).

Compiling and Running the Weather App in Visual Studio

In the Visual Studio solution where you added the library, add to your project the files weather.c and cJSON.c from the weather subfolder of this chapter's examples folder. Specify the command-line arguments as follows:

- Right-click the project name in the Solution Explorer and select Properties.
- Expand Configuration Properties and select Debugging.
- Enter the arguments in the textbox to the right of **Command Arguments**.
- Build and run your project.

Challenge: Create Your Own Mashup

We introduced mashups in Section 1.11.3. After studying our libcurl-and-webservices-based weather application's code, including the extensive comments, as a challenge exercise attempt your first mashup. Web mashups typically combine capabilities from two or more complementary web services. For many popular mashups one of those is a mapping service, such as Google Maps or Microsoft's Bing Maps, but there are many mashup possibilities that do not use mapping.

To build a web-services mashup, you typically need:

- two or more complementary web services, which when you mash them up will help you produce a valuable new application.
- to be able to send a request from your C program to a web service, as you learned how to do with libcurl in this case study.
- to be able to receive results back from that web service in a form (typically JSON) that your C program can understand.

The web-services directory ProgrammableWeb

https://programmableweb.com/

lists nearly 24,000 web services and 8,000 mashups. They also provide "how-to" guides and sample code for working with web services and creating your own mashups. According to their website, some of the most widely used web services are Google Maps and others from Facebook, Twitter and YouTube.

Familiarize yourself with ProgrammableWeb. Look at lots of the web services they describe, focusing on free ones—it's common for web-service providers to offer some free services and some paid. Read the ProgrammableWeb "how-to" guides on creating your own mashups. For inspiration, glance through some of the 8,000 mashups they list on their site. Try to find two complementary free web services from which you can create a valuable mashup, then build that mashup.

Data Structures



Objectives

In this chapter, you'll:

- Allocate and free memory dynamically for data objects.
- Form linked data structures using pointers, self-referential structures and recursion.
- Create and manipulate linked lists, queues, stacks and binary trees.
- Learn important applications of linked data structures.
- Study Secure C programming recommendations for pointers and dynamic memory allocation.
- Optionally build your own compiler in the exercises.

- **12.1** Introduction
- **12.2** Self-Referential Structures
- 12.3 Dynamic Memory Management
- 12.4 Linked Lists
 - 12.4.1 Function insert
 - 12.4.2 Function delete
 - 12.4.3 Functions is Empty and printList
- 12.5 Stacks
 - 12.5.1 Function push
 - 12.5.2 Function pop
 - 12.5.3 Applications of Stacks

- 12.6 Queues
 - 12.6.1 Function enqueue
 - 12.6.2 Function dequeue
- **12.7** Trees
 - 12.7.1 Function insertNode
 - 12.7.2 Traversals: Functions inOrder, preOrder and postOrder
 - 12.7.3 Duplicate Elimination
 - 12.7.4 Binary Tree Search
 - 12.7.5 Other Binary Tree Operations
- **12.8** Secure C Programming

Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises | Special Section: Building Your Own Compiler

12.1 Introduction

We've studied fixed-size data structures, including one-dimensional arrays, two-dimensional arrays and structs. This chapter introduces dynamic data structures that can grow and shrink at execution time:

- Linked lists are collections of data items "lined up in a row." You can insert and delete items anywhere in a linked list.
- Stacks are important in compilers and operating systems. You can insert and delete items only at one end of a stack, known as its top.
- Queues represent waiting lines. You can insert only at the queue's back and delete only from its front. The back and front are known as the queue's tail and head.
- Binary trees facilitate high-speed searching and sorting of data, efficiently
 eliminating duplicate data items and compiling expressions into machine language.

Each of these data structures has many other interesting applications.

Optional Project: Building Your Own Compiler

We hope you'll attempt the optional major project described in the special section entitled Building Your Own Compiler at the end of the exercises. You've been using a compiler to translate your C programs to machine language so that you could execute them. In this project, you'll build your own compiler. It will read a file of statements written in a simple yet powerful, high-level language. Your compiler will translate these statements into a file of Simpletron Machine Language (SML) instructions. SML is the (Deitel-created) language you learned in Chapter 7's special section, Building Your Own Computer. Your Simpletron Simulator program will then execute the SML program produced by your compiler! This project enables you to exercise most of what you've learned in this book. The special section carefully walks you

through the high-level language's specifications and describes the algorithms for converting each high-level language statement into machine-language instructions. If you enjoy challenges, you might attempt the many enhancements to both the compiler and the Simpletron Simulator we suggest in the exercises.

Self Check

I (Fill-In) Which data structure is described by "facilitates high-speed searching and sorting of data, efficiently eliminating duplicate data items and compiling expressions into machine language"? ______.

Answer: Binary tree.

2 (*Fill-In*) Which data structure is described by "a collection of data items lined up in a row—insertions and deletions are made anywhere in the data structure"?

Answer: Linked list.

3 *(Fill-In)* Which dynamic data structure is described by "You can insert and delete items only at one end, the top"? _____.

Answer: Stack.

12.2 Self-Referential Structures

A self-referential structure contains a pointer member that points to a structure of the *same* structure type. For example, the following definition creates the type, struct node:

```
struct node {
   int data;
   struct node *nextPtr;
};
```

Our struct node has two members—integer member data and pointer member nextPtr. The nextPtr points to another struct node. This structure has the same type as the one we're defining, hence the term self-referential structure. Member nextPtr is a link—it can be used to link a struct node to another struct node. We link self-referential structure objects to form lists, queues, stacks and trees.

The following diagram illustrates two self-referential structure objects linked together to form a list:



The slash¹ in the last node represents a NULL pointer, which indicates that the node does not point to another node. A NULL pointer indicates the end of a data structure. Not setting the last node's link to NULL can lead to runtime errors.



^{1.} The slash is only for illustration purposes. It does not correspond to the C's backslash character.



Self Check

I (*Fill-In*) What should replace ??? in the following code to make this a self-referential struct? _____.

```
struct node {
   int data;
   ??? *nextPtr;
};
```

Answer: struct node.

2 (*Fill-In*) A _____ pointer indicates the end of a data structure.

Answer: NULL.

3 (*Fill-In*) Self-referential structures can be ______ together to form data structures such as lists, queues, stacks and trees.

Answer: linked.

12.3 Dynamic Memory Management

Creating and maintaining dynamic data structures that grow and shrink at execution time requires dynamic memory management, which has two components:

- obtaining more memory at execution time to hold new nodes, and
- releasing memory no longer needed.

The function malloc, the function free and the operator size of are essential to dynamic memory management.

The malloc Function

To request memory at execution time, pass to the function malloc the number of bytes to allocate. If successful, malloc returns a void * pointer to the allocated memory. Recall that a void * pointer may be assigned to a variable of *any* pointer type.

Function malloc most commonly is used with sizeof. For example, the following statement determines a struct node object's size in bytes with sizeof(struct node), allocates a new area in memory of that number of bytes and stores a pointer to the allocated memory in newPtr:

```
newPtr = malloc(sizeof(struct node));
```



The memory is not guaranteed to be initialized, though many implementations initialize it for security. If no memory is available, malloc returns NULL. Always test for a NULL pointer before accessing the dynamically allocated memory to avoid runtime errors that might crash your program.

The free Function

When you no longer need a block of dynamically allocated memory, return it to the system immediately by calling the **free** function to deallocate the memory. This

returns it to the system for potential reallocation in the future. To free the memory from the preceding malloc call, use the statement

```
free(newPtr);
```

After deallocating memory, set the pointer to NULL. This prevents accidentally referring to that memory, which may have already been allocated for another purpose.

Not freeing dynamically allocated memory when it's no longer needed can cause (X) ERR the system to run out of memory prematurely. This is sometimes called a "memory leak." Referring to memory that has been freed is an error that typically causes a program to crash. Freeing memory that you did not allocate dynamically with malloc is (X) ERR an error.





Functions calloc and realloc

C also provides the functions calloc and realloc for creating and modifying the size of dynamic arrays. Section 15.8 discusses these functions.



Self Check

I (True/False) Function malloc takes as an argument the number of bytes to be allocated and returns a NULL pointer.

Answer: False. Actually, malloc returns a void * pointer with the allocated memory's address or returns a NULL pointer if the memory could not be allocated.

2 (*Discussion*) Describe precisely what the following statement does:

```
newPtr = malloc(sizeof(struct node));
```

Answer: The statement evaluates sizeof(struct node) to determine the object's size in bytes, allocates a new area in memory of that number of bytes and stores a pointer to the allocated memory in newPtr.

3 (Discussion) Write a statement that frees the memory that was dynamically allocated to newPtr by malloc.

Answer: free(newPtr);

4 (Fill-In) Not freeing dynamically allocated memory when it's no longer needed can cause the system to run out of memory. This is sometimes called a(n) _____. **Answer:** memory leak.

12.4 Linked Lists

A linked list is a linear collection of self-referential struct objects, called nodes, connected by pointer links—hence the term "linked" list. You access a linked list via a pointer to its first node and access subsequent nodes via the nodes' pointer link members. You dynamically store data in a linked list by creating each node as necessary. A node may contain any type of data, including other struct objects. Stacks and queues are also linear data structures. You'll soon see that these are constrained versions of linked lists.

Arrays vs. Linked Lists

You can store lists of data in arrays, but linked lists provide several advantages:

- A linked list is appropriate when the number of data items is unpredictable. A linked list is dynamic, so its length can increase or decrease as necessary. Arrays are fixed-size data structures (though Section 15.8 shows how to dynamically allocate and reallocate arrays).
- PERF 💖
- An array can be declared to contain more elements than the number of data items expected, but this can waste memory. Using linked lists and dynamic memory allocation for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that the pointers in a list's nodes require additional memory. Also, dynamic memory allocation incurs the overhead of function calls.
- Fixed-size arrays can become full. Linked lists become full only when the system has insufficient *memory* to satisfy dynamic storage-allocation requests.
- Linked lists can be maintained in sorted order by inserting each new element at the appropriate point in the list. Inserting into and deleting from a sorted array can be time-consuming. All elements following the inserted or deleted element must be shifted appropriately.

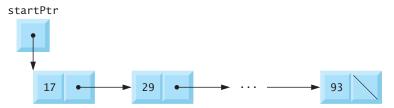
PERF 🐉

Arrays Are Faster for Direct Element Access

PERF SO On the other hand, array elements are stored contiguously in memory. This allows immediate access to any array element—any element's address can be calculated directly based on its position relative to the beginning of the array. Linked lists do not afford such immediate access to their elements.

Illustrating a Linked List

Linked-list nodes are not guaranteed to be stored contiguously in memory. Logically, however, the nodes appear to be contiguous. The following diagram illustrates a linked list with several nodes.



Implementing a Linked List

Figure 12.1 manipulates a list of characters. You can insert a character in the list in alphabetical order (function insert) or delete a character from the list (function delete). A detailed discussion of the program follows. We've split this program's code for discussion purposes—the output is shown at the end of the first code table below.

```
// fig12_01.c
    // Inserting and deleting nodes in a list
 3
   #include <stdio.h>
   #include <stdlib.h>
 4
 5
    // self-referential structure
 6
 7
    struct listNode {
       char data; // each listNode contains a character
 9
       struct listNode *nextPtr; // pointer to next node
10
    };
11
12
    typedef struct listNode ListNode; // synonym for struct listNode
    typedef ListNode *ListNodePtr; // synonym for ListNode*
13
14
15
   // prototypes
   void insert(ListNodePtr *sPtr, char value);
16
17
    char delete(ListNodePtr *sPtr, char value);
    int isEmpty(ListNodePtr sPtr);
19
    void printList(ListNodePtr currentPtr);
20
    void instructions(void);
21
22
    int main(void) {
       ListNodePtr startPtr = NULL; // initially there are no nodes
23
24
       char item = '\0'; // char entered by user
25
26
       instructions(); // display the menu
       printf("%s", "? ");
27
       int choice = 0; // user's choice
28
29
       scanf("%d", &choice);
30
       // loop while user does not choose 3
31
32
       while (choice != 3) {
          switch (choice) {
33
34
             case 1: // insert an element
                printf("%s", "Enter a character: ");
35
                scanf("\n%c", &item);
36
37
                insert(&startPtr, item); // insert item in list
38
                printList(startPtr);
39
                break;
40
             case 2: // delete an element
41
                if (!isEmpty(startPtr)) { // if list is not empty
42
                   printf("%s", "Enter character to be deleted: ");
                   scanf("\n%c", &item);
43
44
45
                   // if character is found, remove it
46
                   if (delete(&startPtr, item)) { // remove item
47
                       printf("%c deleted.\n", item);
48
                      printList(startPtr);
49
                   }
50
                   else {
                       printf("%c not found.\n\n", item);
51
52
                   }
53
                }
```

Fig. 12.1 Inserting and deleting nodes in a list. (Part 1 of 3.)

```
54
                 else {
55
                    puts("List is empty.\n");
56
57
58
                 break;
59
              default:
                 puts("Invalid choice.\n");
60
                 instructions();
61
62
                 break;
63
          }
64
          printf("%s", "? ");
65
          scanf("%d", &choice);
66
67
       } // end while
68
69
       puts("End of run.");
70 }
71
72 // display program instructions to user
73 void instructions(void) {
74
       puts("Enter your choice:\n"
75
              1 to insert an element into the list.\n"
76
              2 to delete an element from the list.\n"
          " 3 to end.");
77
78 }
79
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 1
Enter a character: B
The list is:
B --> NULL
? 1
Enter a character: A
The list is:
A --> B --> NULL
? 1
Enter a character: C
The list is:
A \longrightarrow B \longrightarrow C \longrightarrow NULL
Enter character to be deleted: D
D not found.
```

Fig. 12.1 | Inserting and deleting nodes in a list. (Part 2 of 3.)

```
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL
? 2
Enter character to be deleted: C
C deleted.
The list is:
A --> NULL
Enter character to be deleted: A
A deleted.
List is empty.
Invalid choice.
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 3
End of run.
```

Fig. 12.1 | Inserting and deleting nodes in a list. (Part 3 of 3.)

Lines 7–10 define the self-referential structure struct listNode, which we use to build this example's linked list. Lines 12 and 13 define typedefs that we use to make the code more readable. The name ListNode represents a struct listNode object, and the name ListNodePtr represents a pointer to a struct listNode object. The main function enables you to insert characters in the list (lines 34–39), delete items from the list (lines 40–58) or terminate the program. Initially, startPtr (line 23) is set to NULL to indicate an empty list. The program's primary linked-list functions are insert (Section 12.4.1) and delete (Section 12.4.2).

12.4.1 Function insert

For this example, we insert characters in the list in alphabetical order. Function insert (lines 81–110) receives as an argument the *address of the pointer to the list's first node* and a character to insert. This enables insert to modify the caller's pointer to the list's first node to point to a new first node when a data item is placed at the front of the list. So we pass the *pointer* by reference. Passing a pointer's address creates a pointer to a pointer—this is sometimes called double indirection. This is a complex notion that requires careful programming.

```
// insert a new value into the list in sorted order
80
    void insert(ListNodePtr *sPtr, char value) {
81
       ListNodePtr newPtr = malloc(sizeof(ListNode)); // create node
82
83
84
       if (newPtr != NULL) { // is space available?
          newPtr->data = value; // place value in node
85
          newPtr->nextPtr = NULL; // node does not link to another node
86
87
88
          ListNodePtr previousPtr = NULL;
          ListNodePtr currentPtr = *sPtr;
89
90
          // loop to find the correct location in the list
91
92
          while (currentPtr != NULL && value > currentPtr->data) {
93
             previousPtr = currentPtr; // walk to ...
             currentPtr = currentPtr->nextPtr; // ... next node
94
95
          }
96
97
          // insert new node at beginning of list
          if (previousPtr == NULL) {
98
             newPtr->nextPtr = *sPtr;
99
100
             *sPtr = newPtr;
          }
101
          else { // insert new node between previousPtr and currentPtr
102
             previousPtr->nextPtr = newPtr;
103
             newPtr->nextPtr = currentPtr;
104
          }
105
106
       }
       else {
107
108
          printf("%c not inserted. No memory available.\n", value);
109
       }
110 }
111
```

The insert function performs the following steps:

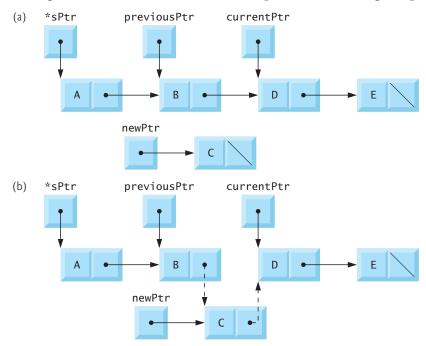
- 1. Call malloc to create a new node and assign newPtr the allocated memory's address (line 82).
- 2. If the memory was allocated, assign the character to insert to newPtr->data (line 85), and assign NULL to newPtr->nextPtr (line 86). Always assign NULL to a new node's link member. Pointers should be initialized before they're used.
- **3.** We'll use pointers previousPtr and currentPtr to store the locations of the node *preceding* and *after* the insertion point, respectively. Initialize previousPtr to NULL (line 88) and currentPtr to *sPtr (line 89)—the first node's address.
- 4. Locate the new value's insertion point. While currentPtr is not NULL and the value to insert is greater than currentPtr->data (line 92), assign currentPtr to previousPtr (line 93), then advance currentPtr to the list's next node (line 94).
- 5. Insert the new value in the list. If previousPtr is NULL (line 98), insert the new node as the *first* in the list (lines 99–100). Assign *sPtr to newPtr->nextPtr (the new node's link points to the former first node), and assign newPtr to *sPtr so startPtr in main points to the new first node. Otherwise, insert the new node

in place (lines 103–104). Assign newPtr to previousPtr->nextPtr (the previous node points to the new node) and assign currentPtr to newPtr->nextPtr (the *new* node link points to the *current* node).

For simplicity, we implemented function insert (and other similar functions in this chapter) with a void return type. Function malloc may *fail* to allocate the requested memory. In this case, it would be better for our insert function to return a status that indicates whether the operation was successful.

Illustrating an Insert

The following diagram illustrates inserting a node containing 'C' into an ordered list. Part (a) shows the list and the new node just before the insertion. Part (b) shows the result of inserting the new node. Dotted arrows represent the reassigned pointers.



12.4.2 Function delete

Function delete (lines 113–141) receives the address of the pointer to the list's first node and a character to delete.

```
II2 // delete a list element
II3 char delete(ListNodePtr *sPtr, char value) {
114
       // delete first node if a match is found
115
       if (value == (*sPtr)->data) {
          ListNodePtr tempPtr = *sPtr; // hold onto node being removed
116
117
          *sPtr = (*sPtr)->nextPtr; // de-thread the node
          free(tempPtr); // free the de-threaded node
118
119
          return value;
120
       }
```

```
121
       else {
          ListNodePtr previousPtr = *sPtr;
122
          ListNodePtr currentPtr = (*sPtr)->nextPtr;
123
124
125
          // loop to find the correct location in the list
          while (currentPtr != NULL && currentPtr->data != value) {
126
             previousPtr = currentPtr; // walk to ...
127
             currentPtr = currentPtr->nextPtr; // ... next node
128
          }
129
130
          // delete node at currentPtr
131
          if (currentPtr != NULL) {
132
133
             ListNodePtr tempPtr = currentPtr;
             previousPtr->nextPtr = currentPtr->nextPtr;
134
135
             free(tempPtr);
136
             return value;
137
          }
       }
138
139
140
       return '\0';
141 }
142
```

The delete function performs the following steps:

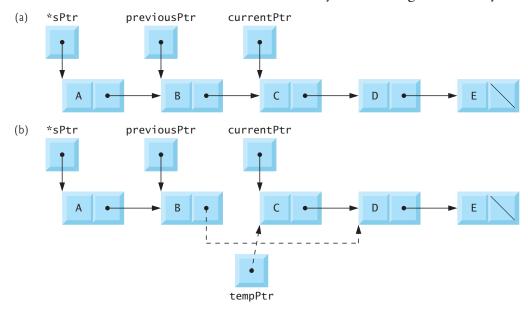
- 1. If the character to delete matches the first node's character (line 115), we must remove the first node. Assign *sPtr to tempPtr, which we'll use to free the node's memory. Assign (*sPtr)->nextPtr to *sPtr, so that startPtr in main now points to what was previously the list's second node. Call free to deallocate the memory pointed to by tempPtr. Return the character that was deleted.
- **2.** Otherwise, initialize previousPtr with *sPtr and initialize currentPtr with (*sPtr)->nextPtr (lines 122–123) to advance to the second node.
- 3. Locate the character to delete. While currentPtr is not NULL and the value to delete is not equal to currentPtr->data (line 126), assign currentPtr to pre-viousPtr (line 127) and assign currentPtr->nextPtr to currentPtr (line 128) to advance to the list's next node.
- 4. If currentPtr is not NULL (line 132), the character is in the list. Assign currentPtr to tempPtr (line 133), which we'll use to deallocate the node. Assign currentPtr->nextPtr to previousPtr->nextPtr (line 134) to connect the node before and the node after the one being removed. Free the node pointed to by tempPtr (line 135), then return the deleted character (line 136).

If nothing has been returned yet, line 140 returns the null character ('\0') to signify that the character was not found in the list.

Illustrating a Delete

The following diagram illustrates deleting the node containing 'C' from a linked list. Part (a) shows the linked list before the deletion. Part (b) shows the link reassignments. Pointer tempPtr is used to free the memory allocated to the node that stores

'C'. Note that in lines 118 and 135, we free tempPtr. Previously, we recommended setting a freed pointer to NULL. We do not do that here because tempPtr is a local automatic variable, and the function returns immediately after freeing the memory.



12.4.3 Functions is Empty and printList

Function is Empty (lines 144–146) is a predicate function—it does not alter the list in any way. Rather, is Empty determines whether the list is empty—that is, the pointer to the first node is NULL. If the list is empty, is Empty returns 1; otherwise, it returns 0.

```
143 // return 1 if the list is empty, 0 otherwise
144 int isEmpty(ListNodePtr sPtr) {
145
       return sPtr == NULL;
146 }
147
   // print the list
    void printList(ListNodePtr currentPtr) {
150
       // if list is empty
151
       if (isEmpty(currentPtr)) {
152
          puts("List is empty.\n");
153
       }
154
       else {
155
          puts("The list is:");
156
          // while not the end of the list
157
          while (currentPtr != NULL) {
158
             printf("%c --> ", currentPtr->data);
159
             currentPtr = currentPtr->nextPtr;
160
161
162
          puts("NULL\n");
163
164
165 }
```

Function printList (lines 149–165) prints a list. The function's currentPtr parameter receives a pointer to the list's first node. The function first determines whether the list is *empty* (lines 151–153) and, if so, prints "List is empty." and terminates. Otherwise, lines 155-163 print the list's data. While currentPtr is not NULL, line 159 prints the value in currentPtr->data, and line 160 assigns currentPtr->nextPtr to currentPtr to advance to the next node. If the link in the last node of the list is not NULL, the printing algorithm will try to print past the end of the ERR (X) list, which is a logic error. This printing algorithm is identical for linked lists, stacks and queues.

List-Based Recursion Exercises

Exercise 12.17 asks you to implement a recursive function that prints a list backward. Exercise 12.18 asks you to implement a recursive function that searches a linked list for a particular data item.

Self Check

- (Multiple Choice) Which of the following statements a), b) or c) is false?
 - a) A linked list is a linear collection of self-referential structures, called nodes, connected by pointer links—hence, the term "linked" list.
 - b) A linked list is accessed via a pointer to the first node of the list.
 - c) Subsequent nodes in a linked list are accessed via the link pointer member stored in each node.
 - d) All of the above statements are true.

Answer: d.

2 (*True/False*) A linked list is appropriate when the number of data items to store in the data structure is unpredictable.

Answer: True.

3 (True/False) The elements in a linked list are stored contiguously in memory. This allows immediate access to any elements because its address can be calculated directly, based on its position relative to the beginning of the linked list. Arrays do not afford such immediate access to their elements.

Answer: False. Actually, the reverse is true. An array's elements are stored contiguously in memory and support immediate access by position. Linked lists do not afford such immediate access to their elements.

4 (Fill-In) Passing into a function the address of the pointer to a linked list's first node creates a pointer to a pointer. This is often called double _____. **Answer:** indirection.

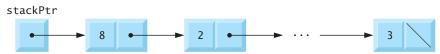
12.5 Stacks

A stack can be implemented as a constrained version of a linked list. You add new nodes and remove existing ones only at the top. For this reason, a stack is referred to

as a last-in, first-out (LIFO) data structure. You access a stack via a pointer to its top element. The link member in the stack's last node is set to NULL to indicate the stack's bottom. Not terminating the stack with NULL can lead to runtime errors.



The following diagram illustrates a stack with several nodes—stackPtr points to the stack's top element. We represent stacks and linked lists in these figures identically. The difference between them is that insertions and deletions may occur anywhere in a linked list, but only at the top of a stack.



Primary Stack Operations

A stack's primary functions are push and pop. Function push creates a new node and places it on top of the stack. Function pop removes a node from the stack's top, frees that node's memory and returns the popped value.

Implementing a Stack

Figure 12.2 implements a simple stack of integers. The program allows you to push a value onto the stack (function push), pop a value off the stack (function pop) and terminate the program. Lines 7–10 define struct stackNode, which we'll use to represent the stack's nodes. As in Fig. 12.1, we use typedefs (lines 12–13) to make the code more readable. Initially, stackPtr (line 23) is set to NULL to indicate an empty stack. Much of this application's logic is similar to Fig. 12.1, so we focus on the differences here.

```
I // fig12_02.c
 2 // A simple stack program
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5
    // self-referential structure
 7
    struct stackNode {
       int data; // define data as an int
 8
 9
       struct stackNode *nextPtr; // stackNode pointer
10
    };
11
    typedef struct stackNode StackNode; // synonym for struct stackNode
12
    typedef StackNode *StackNodePtr; // synonym for StackNode*
14
15
    // prototypes
   void push(StackNodePtr *topPtr, int info);
int pop(StackNodePtr *topPtr);
18 int isEmpty(StackNodePtr topPtr);
19 void printStack(StackNodePtr currentPtr);
20
   void instructions(void);
21
```

Fig. 12.2 A simple stack program. (Part 1 of 4.)

```
22
    int main(void) {
23
       StackNodePtr stackPtr = NULL; // points to stack top
24
       int value = 0; // int input by user
25
26
       instructions(); // display the menu
       printf("%s", "? ");
27
       int choice = 0; // user's menu choice
28
       scanf("%d", &choice);
29
30
31
       // while user does not enter 3
       while (choice != 3) {
32
33
          switch (choice) {
34
             case 1: // push value onto stack
                 printf("%s", "Enter an integer: ");
scanf("%d", &value);
35
36
37
                 push(&stackPtr, value);
38
                 printStack(stackPtr);
39
             case 2: // pop value off stack
40
                 // if stack is not empty
41
42
                 if (!isEmpty(stackPtr)) {
43
                    printf("The popped value is %d.\n", pop(&stackPtr));
44
                 }
45
46
                 printStack(stackPtr);
47
                 break:
48
             default:
                 puts("Invalid choice.\n");
49
50
                 instructions();
51
                 break;
52
          }
53
          printf("%s", "? ");
54
55
          scanf("%d", &choice);
56
       }
57
58
       puts("End of run.");
59
   }
60
61
   // display program instructions to user
62 void instructions(void) {
63
       puts("Enter choice:\n"
64
          "1 to push a value on the stack\n"
65
          "2 to pop a value off the stack\n"
66
          "3 to end program");
67
   }
68
69
    // insert a node at the stack top
70 void push(StackNodePtr *topPtr, int info) {
71
       StackNodePtr newPtr = malloc(sizeof(StackNode));
72
```

Fig. 12.2 A simple stack program. (Part 2 of 4.)

```
if (newPtr != NULL) {
74
75
          newPtr->data = info;
76
          newPtr->nextPtr = *topPtr;
77
          *topPtr = newPtr;
78
       }
79
       else { // no space available
80
          printf("%d not inserted. No memory available.\n", info);
81
    }
82
83
84
    // remove a node from the stack top
85
    int pop(StackNodePtr *topPtr) {
86
       StackNodePtr tempPtr = *topPtr;
87
       int popValue = (*topPtr)->data;
88
       *topPtr = (*topPtr)->nextPtr;
89
       free(tempPtr);
90
       return popValue;
    }
91
92
93
   // print the stack
94
    void printStack(StackNodePtr currentPtr) {
95
       if (currentPtr == NULL) { // if stack is empty
96
          puts("The stack is empty.\n");
       }
97
98
       else {
99
          puts("The stack is:");
100
101
          while (currentPtr != NULL) { // while not the end of the stack
102
             printf("%d --> ", currentPtr->data);
103
             currentPtr = currentPtr->nextPtr;
104
          }
105
106
          puts("NULL\n");
       }
107
108 }
109
IIO // return 1 if the stack is empty, 0 otherwise
III int isEmpty(StackNodePtr topPtr) {
112
       return topPtr == NULL;
113 }
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL
```

Fig. 12.2 | A simple stack program. (Part 3 of 4.)

73

// insert the node at stack top

```
? 1
Enter an integer: 6
The stack is:
6 --> 5 --> NULL
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL
The popped value is 4.
The stack is:
6 --> 5 --> NULL
The popped value is 6.
The stack is:
5 --> NULL
The popped value is 5.
The stack is empty.
The stack is empty.
? 4
Invalid choice.
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

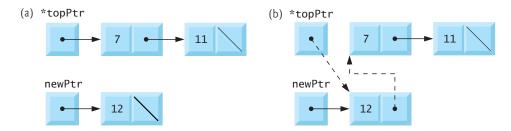
Fig. 12.2 | A simple stack program. (Part 4 of 4.)

12.5.1 Function push

Function push (lines 70–82) places a new node onto the stack using the following steps:

- 1. Call malloc to create a new node, then assign the allocated memory's address to newPtr (line 71).
- 2. Assign to newPtr->data the value to push onto the stack (line 75) and assign *topPtr (the pointer to the stack's top) to newPtr->nextPtr (line 76). The link member of the new top node now points to the previous top node.
- 3. Assign newPtr to *topPtr (line 77)—this modifies stackPtr in main to point to the new stack top.

The following diagram illustrates a push operation. Part (a) shows the stack and the new node before the push operation inserts the new node at the stack's top—*topPtr represents stackPtr in main. The dotted arrows in Part (b) illustrate *Steps 2* and *3* of the preceding discussion, which insert the node containing 12 at the top.

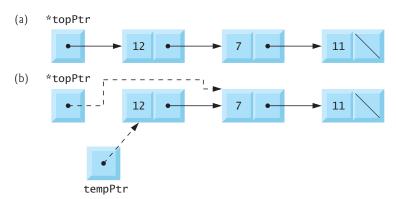


12.5.2 Function pop

Function pop (lines 85–91) removes the stack's top node. Function main determines whether the stack is empty before calling pop. The pop operation consists of five steps:

- 1. Assign *topPtr to tempPtr (line 86), which will be used to free the node's memory.
- 2. Assign (*topPtr)->data to popValue (line 87) to save the top node's value so we can return it.
- 3. Assign (*topPtr)->nextPtr to *topPtr (line 88) so that stackPtr in main now points to what was previously the stack's second element (or NULL if there were no other elements).
- **4.** Free the memory pointed to by tempPtr (line 89).
- **5.** Return popValue to the caller (line 90).

The following diagram illustrates a pop operation. Part (a) shows the stack before removing the node containing 12—*topPtr represents stackPtr in main. Part (b) shows tempPtr pointing to the node being popped and *topPtr pointing to the new top node. Then we can free the memory to which tempPtr points.



12.5.3 Applications of Stacks

Stacks have many interesting applications. For example, whenever a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack (Section 5.7). In a series of function calls, the successive return addresses are pushed onto the stack in last-in, first-out order so that each function can return to its caller. Stacks support recursive function calls in the same manner as conventional nonrecursive calls.

614 Chapter 12 Data Structures

Stacks contain the space created for automatic local variables on each invocation of a function. When the function returns to its caller, the space for that function's automatic variables is popped off the stack, and these variables are no longer known to the program. Stacks also are sometimes used by compilers in the process of evaluating expressions and generating machine-language code. The exercises explore several stack applications.

✓ Self Check

- [(Multiple Choice) Which of the following statements is false?
 - a) A stack must be implemented as a constrained version of a linked list.
 - b) New nodes are added to and removed from a stack only at the top.
 - c) A stack is a last-in, first-out (LIFO) data structure.
 - d) A stack is referenced via a pointer to its top element.

Answer: (a) is *false*. Actually, a stack *can* be implemented as a constrained version of a linked list, but it need not be.

- **2** (Multiple Choice) Which of the following statements a), b) or c) is false?
 - a) When a function is called, it must know how to return to its caller, so the called function's return address is pushed onto the stack.
 - b) In a series of function calls, the successive return addresses are pushed onto the stack in last-in, first-out order so that each function can return to its caller.
 - c) Stacks support recursive function calls in the same manner as conventional nonrecursive calls.
 - d) All of the above statements are true.

Answer: (a) is false. Actually, the caller's return address is pushed onto the stack.

12.6 Queues

A queue is similar to a checkout line in a grocery store:

- The first person in line receives service first.
- Other customers enter the line only at the end and wait for service.

You remove queue nodes only from its head (front) and insert nodes only at its tail (back). For this reason, a queue is referred to as a first-in, first-out (FIFO) data structure. The insert and remove operations are known as enqueue (pronounced "en-cue") and dequeue (pronounced "dee-cue"), respectively.

Queue Applications

Queues have many applications in computer systems:

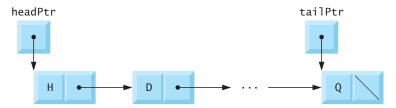
• For computers that have only a single processor, only one user at a time may be serviced. Entries for the other users are placed in a queue. Each entry gradually advances to the front of the queue as users receive service. The entry at the front of the queue is the next to receive service.

- Similarly, for today's multicore systems, there could be more programs running than there are processors. The programs not currently running are placed in a queue until a currently busy processor becomes available. In Appendix C, we discuss multithreading. When a program's work is divided into multiple threads capable of executing in parallel, there could be more threads than processors. The threads not currently running need to wait in a queue.
- Queues also support print spooling. An office may have only one printer. Many users can send documents to print at a given time. When the printer is busy, additional documents are spooled to memory or secondary storage, just as sewing thread is wrapped around a spool until it's needed. The documents wait in a queue until the printer becomes available.
- Information packets traveling over computer networks, like the Internet, also wait in queues. Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to its final destination. The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.

Illustrating a Queue

The following diagram illustrates a queue with several nodes. Note the separate pointers to the queue's head and tail. As with lists and stacks, not setting the link in the (\times) ERR queue's last node to NULL can lead to logic errors.





Implementing a Queue

Figure 12.3 performs queue manipulations. The program provides options to insert a node in the queue (function enqueue), remove a node from the queue (function **dequeue**) and terminate the program. Lines 7–10 define struct queueNode, which we'll use to represent the queue's nodes. Again, we use typedefs (lines 12–13) to make the code more readable. Much of the logic in this application is similar to our list and stack examples, so we focus on the differences here. Initially, headPtr and tailPtr (lines 23–24) are both NULL to indicate an empty queue.

```
// fig12_03.c
  // Operating and maintaining a queue
3 #include <stdio.h>
  #include <stdlib.h>
4
5
```

Operating and maintaining a queue. (Part 1 of 5.)

```
// self-referential structure
 7
    struct queueNode {
       char data; // define data as a char
 8
 9
       struct queueNode *nextPtr; // queueNode pointer
10
    };
11
12
    typedef struct queueNode QueueNode;
    typedef QueueNode *QueueNodePtr;
13
14
15
   // function prototypes
16
   void printQueue(QueueNodePtr currentPtr);
   int isEmpty(QueueNodePtr headPtr);
17
   void enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value);
18
19
    char dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr);
20
    void instructions(void);
21
22
    int main(void) {
23
       QueueNodePtr headPtr = NULL; // initialize headPtr
24
       QueueNodePtr tailPtr = NULL; // initialize tailPtr
25
       char item = '\0'; // char input by user
26
27
       instructions(); // display the menu
       printf("%s", "? ");
28
29
       int choice = 0; // user's menu choice
       scanf("%d", &choice);
30
31
32
       // while user does not enter 3
33
       while (choice != 3) {
          switch(choice) {
34
35
             case 1: // enqueue value
                 printf("%s", "Enter a character: ");
scanf("\n%c", &item);
36
37
38
                 enqueue(&headPtr, &tailPtr, item);
39
                 printQueue(headPtr);
40
                 break:
             case 2: // dequeue value
41
42
                 // if queue is not empty
43
                 if (!isEmpty(headPtr)) {
44
                    item = dequeue(&headPtr, &tailPtr);
                    printf("%c has been dequeued.\n", item);
45
                 }
46
47
48
                 printQueue(headPtr);
49
                 break:
50
             default:
51
                 puts("Invalid choice.\n");
52
                 instructions();
53
                 break;
54
          }
55
          printf("%s", "? ");
56
57
          scanf("%d", &choice);
58
       }
```

Fig. 12.3 Operating and maintaining a queue. (Part 2 of 5.)

```
59
60
       puts("End of run.");
    }
61
62
63
    // display program instructions to user
    void instructions(void) {
       printf ("Enter your choice:\n"
65
                    1 to add an item to the queue\n"
66
67
                    2 to remove an item from the queue\n"
68
                   3 to end\n");
69
    }
70
    // insert a node at queue tail
71
    void enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value) {
72
73
       QueueNodePtr newPtr = malloc(sizeof(QueueNode));
74
75
       if (newPtr != NULL) { // is space available?
76
          newPtr->data = value;
77
          newPtr->nextPtr = NULL;
78
79
          // if empty, insert node at head
          if (isEmpty(*headPtr)) {
80
             *headPtr = newPtr;
81
          }
82
          else {
83
84
             (*tailPtr)->nextPtr = newPtr;
85
86
87
          *tailPtr = newPtr;
88
       }
89
       else {
90
          printf("%c not inserted. No memory available.\n", value);
91
92
    }
93
    // remove node from queue head
95
    char dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr) {
96
       char value = (*headPtr)->data;
       QueueNodePtr tempPtr = *headPtr;
97
       *headPtr = (*headPtr)->nextPtr;
98
99
100
       // if queue is empty
101
       if (*headPtr == NULL) {
          *tailPtr = NULL;
102
103
104
105
       free(tempPtr);
       return value;
106
107 }
108
```

Fig. 12.3 | Operating and maintaining a queue. (Part 3 of 5.)

```
109 // return 1 if the queue is empty, 0 otherwise
110 int isEmpty(QueueNodePtr headPtr) {
       return headPtr == NULL;
111
112 }
113
114 // print the queue
II5 void printQueue(QueueNodePtr currentPtr) {
       if (currentPtr == NULL) { // if queue is empty
          puts("Queue is empty.\n");
117
       }
118
       else {
119
          puts("The queue is:");
120
121
122
          while (currentPtr != NULL) { // while not end of queue
             printf("%c --> ", currentPtr->data);
123
124
             currentPtr = currentPtr->nextPtr;
          }
125
126
127
          puts("NULL\n");
128
129 }
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 1
Enter a character: A
The queue is:
A --> NULL
? 1
Enter a character: B
The queue is:
A --> B --> NULL
Enter a character: C
The queue is:
A --> B --> C --> NULL
? 2
A has been dequeued.
The queue is:
B --> C --> NULL
? 2
B has been dequeued.
The queue is:
C --> NULL
```

Fig. 12.3 Operating and maintaining a queue. (Part 4 of 5.)

```
? 2
C has been dequeued.
Queue is empty.
? 2
Queue is empty.
? 4
Invalid choice.

Enter your choice:
    1 to add an item to the queue
    2 to remove an item from the queue
    3 to end
? 3
End of run.
```

Fig. 12.3 Operating and maintaining a queue. (Part 5 of 5.)

12.6.1 Function enqueue

Function enqueue (lines 72–92) receives three arguments:

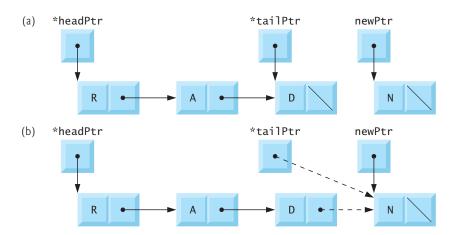
- the address of headPtr—the pointer to the queue's head,
- the address of tailPtr—the pointer to the queue's tail, and
- the value to insert.

The function performs the following steps:

- 1. Line 73 calls malloc to create a new node and assigns the allocated memory location to newPtr.
- 2. If the memory was allocated correctly, lines 76–77 assign the value to insert to newPtr->data, and assign NULL to newPtr->nextPtr.
- 3. If the queue is empty (line 80), line 81 assigns newPtr to *headPtr because the new node is both the queue's head and tail; otherwise, line 84 assigns newPtr to (*tai1Ptr)->nextPtr because the new node is the new tail node.
- **4.** Line 87 assigns newPtr to *tailPtr to update the tail pointer to the new tail node.

Illustrating an enqueue Operation

The following diagram illustrates an enqueue operation. Part (a) shows main's headPtr and tailPtr and enqueue's new node before the operation. The dotted arrows in Part (b) illustrate *Steps 3* and 4 in the preceding discussion, which add the new node to the tail of a non-empty queue.



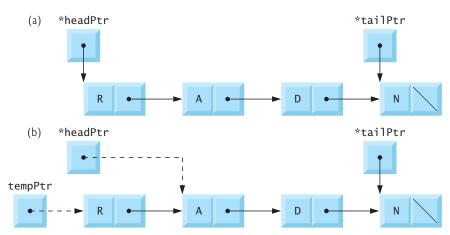
12.6.2 Function dequeue

Function dequeue (lines 95–107) receives as arguments the addresses of the queue's head and tail pointers and removes the queue's first node. The dequeue operation performs the following steps:

- 1. Line 96 assigns (*headPtr)->data to value to save the data being dequeued.
- 2. Line 97 assigns *headPtr to tempPtr, which will be used to free the memory.
- 3. Line 98 assigns (*headPtr)->nextPtr to *headPtr so that the queue's head pointer in main now points to the new head node.
- 4. Line 101 checks whether *headPtr is NULL. If so, line 102 assigns NULL to *tailPtr because the queue is now empty.
- **5.** Line 105 frees the memory pointed to by tempPtr.
- **6.** Line 106 return value to the caller.

Illustrating a dequeue Operation

The following diagram illustrates function dequeue. Part (a) shows the queue before the dequeue operation—*headPtr and *tailPtr in dequeue are used to modify main's headPtr and tailPtr. Part (b) shows tempPtr pointing to the dequeued node, and main's headPtr updated to point to the queue's new first node.



Self Check

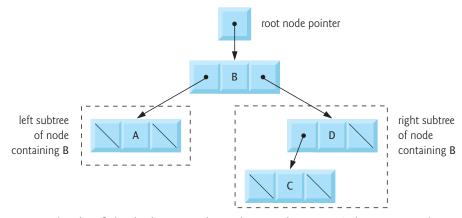
- [(Multiple Choice) Which of the following statements about queues is false?
 - a) A queue is similar to a checkout line in a grocery store—the first person in line receives service first, and other customers enter the line at the end and wait for service.
 - b) Queue nodes are removed only from the queue's head (start) and inserted only at the queue's tail (end).
 - c) A queue is a first-in, last-out (FILO) data structure.
 - d) The insert and remove operations are known as enqueue (pronounced "encue") and dequeue (pronounced "dee-cue"), respectively.

Answer: (c) is false. Actually, a queue is a first-in, first-out (FIFO) data structure.

2 (*Fill-In*) For today's _____ systems, there could be more programs running than there are processors, so the programs not currently running are placed in a queue until a currently busy processor becomes available. **Answer:** multicore.

12.7 Trees

So far, we've presented linear data structures—linked lists, stacks and queues. A tree is a nonlinear, two-dimensional data structure with special properties. Tree nodes contain two or more links. This section discusses binary trees—trees whose nodes contain two links, as in the following diagram:



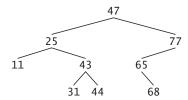
None, one, or both of the links in each node may be NULL. The root node is the first node in a tree. Each link in the root node refers to a child. The left child is the first node in the left subtree, and the right child is the first in the right subtree. A given node's children are called siblings. A node with no children is a leaf node. Not setting a leaf node's links to NULL can lead to runtime errors. Computer scientists typically draw trees with the root node at the top—exactly the opposite of trees in nature.



Binary Search Tree

This section presents a binary search tree containing unique values, which has the characteristic that the values in any *left* subtree are less than the value in its parent

node, and the values in any *right* subtree are greater than the value in its parent node. The figure below illustrates a binary search tree with nine values:



The shape of the binary search tree for a set of data can vary, based on the order in which you insert its values.

Implementing a Binary Search Tree

Figure 12.4 creates a binary search tree and traverses its nodes—that is, it visits each node in the tree to do something with the node values, like display them. We'll traverse the tree three ways—inorder, preorder and postorder. The program generates 10 random numbers and inserts each in the tree, except that we discard duplicate values. Lines 9–13 define struct treeNode, which we'll use to represent the tree's nodes. Again, we use typedefs (lines 15–16) to make the code more readable.

```
| // fig12_04.c
 2 // Creating and traversing a binary tree
 3 // preorder, inorder, and postorder
 4 #include <stdio.h>
 5 #include <stdlib.h>
   #include <time.h>
 7
   // self-referential structure
 8
 9
    struct treeNode {
       struct treeNode *leftPtr; // pointer to left subtree
10
11
       int data; // node value
       struct treeNode *rightPtr; // pointer to right subtree
12
13
    };
14
    typedef struct treeNode TreeNode; // synonym for struct treeNode
15
16
    typedef TreeNode *TreeNodePtr; // synonym for TreeNode *
17
    // prototypes
18
19
   void insertNode(TreeNodePtr *treePtr, int value);
20 void inOrder(TreeNodePtr treePtr);
21
   void preOrder(TreeNodePtr treePtr);
22
   void postOrder(TreeNodePtr treePtr);
24
    int main(void) {
25
       TreeNodePtr rootPtr = NULL; // tree initially empty
26
```

Fig. 12.4 | Creating and traversing a binary tree. (Part 1 of 3.)

```
27
       srand(time(NULL));
28
       puts("The numbers being placed in the tree are:");
29
30
       // insert random values between 0 and 14 in the tree
31
       for (int i = 1; i \le 10; ++i) {
32
          int item = rand() \% 15;
33
          printf("%3d", item);
34
          insertNode(&rootPtr, item);
35
       }
36
37
       // traverse the tree pre0rder
38
       puts("\n\nThe preOrder traversal is:");
39
       preOrder(rootPtr);
40
41
       // traverse the tree inOrder
42
       puts("\n\nThe inOrder traversal is:");
43
       inOrder(rootPtr);
44
45
       // traverse the tree postOrder
46
       puts("\n\nThe postOrder traversal is:");
47
       postOrder(rootPtr);
    }
48
49
    // insert node into tree
50
51
    void insertNode(TreeNodePtr *treePtr, int value) {
52
       if (*treePtr == NULL) { // if tree is empty
53
          *treePtr = malloc(sizeof(TreeNode));
54
55
          if (*treePtr != NULL) { // if memory was allocated, then assign data
              (*treePtr)->data = value;
56
57
              (*treePtr)->leftPtr = NULL;
58
              (*treePtr)->rightPtr = NULL;
59
          }
60
          else {
61
             printf("%d not inserted. No memory available.\n", value);
          }
62
63
       else { // tree is not empty
64
          if (value < (*treePtr)->data) { // value goes in left subtree
65
66
             insertNode(&((*treePtr)->leftPtr), value);
67
          }
68
          else if (value > (*treePtr)->data) { // value goes in right subtree
69
             insertNode(&((*treePtr)->rightPtr), value);
70
71
          else { // duplicate data value ignored
72
             printf("%s", "dup");
73
          }
       }
74
75
    }
76
```

Fig. 12.4 | Creating and traversing a binary tree. (Part 2 of 3.)

```
77
    // begin inorder traversal of tree
78
    void inOrder(TreeNodePtr treePtr) {
       // if tree is not empty, then traverse
79
       if (treePtr != NULL) {
80
81
          inOrder(treePtr->leftPtr);
82
          printf("%3d", treePtr->data);
          inOrder(treePtr->rightPtr);
83
84
85
    }
86
87
    // begin preorder traversal of tree
88
    void preOrder(TreeNodePtr treePtr) {
89
       // if tree is not empty, then traverse
90
       if (treePtr != NULL) {
          printf("%3d", treePtr->data);
91
          preOrder(treePtr->leftPtr);
92
93
          preOrder(treePtr->rightPtr);
94
    }
95
96
97
    // begin postorder traversal of tree
98
    void postOrder(TreeNodePtr treePtr) {
99
       // if tree is not empty, then traverse
100
       if (treePtr != NULL) {
          postOrder(treePtr->leftPtr);
101
102
          postOrder(treePtr->rightPtr);
103
          printf("%3d", treePtr->data);
104
105
The numbers being placed in the tree are:
  6 7 4 12 7dup 2 2dup 5 7dup 11
The preOrder traversal is:
  6 4 2 5 7 12 11
The inOrder traversal is:
  2 4 5 6 7 11 12
The postOrder traversal is:
  2 5 4 11 12 7 6
```

Fig. 12.4 Creating and traversing a binary tree. (Part 3 of 3.)

12.7.1 Function insertNode

The functions in Fig. 12.4 that create a binary search tree and traverse it are recursive. Function insertNode (lines 51–75) receives as arguments the address of the pointer to the tree's root node and an integer to insert. Each new node in a binary search tree initially is inserted as a leaf node. The steps for inserting a new node are as follows:

1. If *treePtr is NULL (line 52), line 53 calls malloc to create a new leaf node and assigns the allocated memory to *treePtr. Lines 56 assigns to (*treePtr)-

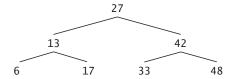
>data the integer to store. Lines 57–58 assign NULL to (*treePtr)->leftPtr and (*treePtr)->rightPtr. Then control returns to the caller—either main or a previous call to insertNode.

- 2. If *treePtr is not NULL and the value to insert is less than (*treePtr)->data, line 66 recursively calls insertNode with the address of (*treePtr)->leftPtr to insert the new node in the left subtree of the node pointed to by treePtr.
- 3. If the value to insert is greater than (*treePtr)-> data, line 69 recursively calls insertNode with the address of (*treePtr)->rightPtr to insert the node in the right subtree of the node pointed to by treePtr.

The recursive steps continue until insertNode finds a NULL pointer, then *Step 1* inserts the new node as a leaf node.

12.7.2 Traversals: Functions in Order, preOrder and postOrder

Functions in Order (lines 78–85), preOrder (lines 88–95) and postOrder (lines 98–105) each receive a pointer to a tree's root node and traverse the tree. Below we describe the traversals and show the result of applying each for the following tree:



inOrder Traversal

The steps for an inOrder traversal are:

- 1. Traverse the *left* subtree in Order (line 81).
- 2. Process the value in the node (line 82).
- **3.** Traverse the *right* subtree in 0 rder (line 83).

This traversal processes each node's value after processing the values in the left subtree. The inOrder traversal of the preceding tree is:

A binary search tree's inOrder traversal processes the nodes in ascending order. Creating a binary search tree actually sorts the data. So, this process is called a binary tree sort.

pre0rder Traversal

The steps for a pre0rder traversal are:

- 1. Process the value in the node (line 91).
- **2.** Traverse the *left* subtree pre0rder (line 92).
- **3.** Traverse the *right* subtree pre0rder (line 93).

This traversal processes each node's value as the node is visited. After processing the value, a pre0rder traversal processes the left subtree's values, then the right subtree's values. The pre0rder traversal of the preceding tree is:

27 13 6 17 42 33 48

postOrder Traversal

The steps for a postOrder traversal are:

- 1. Traverse the *left* subtree postOrder (line 101).
- **2.** Traverse the *right* subtree post0rder (line 102).
- **3.** Process the value in the node (line 103).

This traversal processes each node's value after processing the values of the node's children in both the left and right subtrees. The postOrder traversal of the preceding tree is:

6 17 13 33 48 42 27

12.7.3 Duplicate Elimination

Binary search trees facilitate duplicate elimination. As you insert values to create the tree, a duplicate value will follow the same "go left" or "go right" decisions on each comparison as the original value did. So, the duplicate eventually will be compared with a node in the tree containing the same value. At that point, the duplicate value can be ignored.

12.7.4 Binary Tree Search

Searching for a value that matches a key also is fast. If the tree is tightly packed, each level contains about twice as many elements as the previous level. So, a binary search tree with n elements would have a maximum of $\log_2 n$ levels, and thus requires a maximum of $\log_2 n$ comparisons to find a match or to determine that no match exists. When searching a tightly packed 1,000-element binary search tree, no more than 10 comparisons need to be made because $2^{10} > 1,000$. When searching a tightly packed 1,000,000-element binary search tree, no more than 20 comparisons need to be made because $2^{20} > 1,000,000$.

12.7.5 Other Binary Tree Operations

In the exercises, we present algorithms for several other binary tree operations, such as printing a binary tree in a two-dimensional tree format and performing a level order traversal of a binary tree. The level order traversal visits a tree's nodes row-by-row starting at the root node level. The nodes on each level are visited from left to right. Other binary tree exercises include allowing a binary search tree to contain duplicate values, creating a tree of strings and determining a binary tree's number of levels.

Self Check

- I (*TruelFalse*) Linked lists, stacks, queues and trees are linear data structures. Answer: *False*. Actually, trees are nonlinear, two-dimensional data structures.
- **2** (*Fill-In*) Three popular ways to traverse a binary search tree are ______, preorder and postorder.

Answer: inorder.

- **3** (*Fill-In*) The process of creating a binary search tree actually _____ the data. Answer: sorts.
- **4** *(True/False)* The shape of the binary search tree that corresponds to a set of data is independent of the order in which the values are inserted into the tree.

Answer: *False.* Actually, the shape of the binary search tree that corresponds to a set of data varies, based on the order in which the values are inserted into the tree.

- **5** (*TruelFalse*) A node can be inserted only as a root node in a binary search tree. **Answer:** *False*. Actually, a node can be inserted only as a *leaf* node in a binary search tree.
- **6** (*Fill-In*) A binary search tree facilitates _____. As you insert values to create the tree, identical values follow the same "go left" or "go right" decisions on each comparison as the original value did. An identical value eventually will be compared with a node in the tree containing the same value.

Answer: duplicate elimination.

7 (*TruelFalse*) A tightly packed binary tree with n elements would have a maximum of about $\log_2 n$ levels. Searching such a binary tree requires a maximum of about $\log_2 n$ comparisons to find a match or to determine that no match exists. So, searching a (tightly packed) 1,000,000,000-element binary search tree requires no more than 20 comparisons.

Answer: False. Actually, this requires no more than 30 comparisons.

12.8 Secure C Programming

Chapter 8 of the SEI CERT C Coding Standard is dedicated to memory-management recommendations and rules—many apply this chapter's uses of pointers and dynamic memory management. For more information, visit https://wiki.sei.cmu.edu/.

• MEM01-C/MEM30-C: Pointers should always be initialized with NULL or the address of a valid item in memory. When you use free to deallocate dynamically allocated memory, the pointer passed to free is not assigned a new value, so it still points to the memory location where the dynamically allocated memory used to be. Using such a "dangling" pointer can lead to program crashes and security vulnerabilities. When you free dynamically allocated memory, immediately assign the pointer either NULL or a valid address. We chose not to do this for local pointer variables that immediately go out of scope after a call to free.

- MEM01-C: Undefined behavior occurs when you attempt to use free to deallocate dynamic memory that was already deallocated—this is known as a "double free vulnerability." To ensure that you don't attempt to deallocate the same memory more than once, immediately set a pointer to NULL after the call to free. Freeing a NULL pointer has no effect.
- ERR33-C: Most standard-library functions return values that you can check to determine whether the functions performed their tasks successfully. Function malloc, for example, returns NULL if it's unable to allocate the requested memory. You should always ensure that malloc did not return NULL before attempting to use the pointer that stores malloc's return value.

✓ Self Check

I (True/False) Pointers should always be initialized to NULL.

Answer: *False.* Actually, pointers should always be initialized either to NULL or to the address of a valid item in memory.

2 (*TruelFalse*) To avoid deallocating the same memory twice and causing undefined behavior, set the freed pointer to NULL.

Answer: *True.* Freeing a NULL pointer does not cause undefined behavior.

3 (Fill-In) If malloc can't fulfill a memory-allocation request, it returns _____. Always check for this value before using malloc's pointer return.

Answer: NULL.

Summary

Section 12.1 Introduction

- Dynamic data structures (p. 596) grow and shrink at execution time.
- Linked lists (p. 596) are collections of data items "lined up in a row"—insertions and deletions are made anywhere in a linked list.
- With stacks (p. 596), insertions and deletions are made only at the top (p. 596).
- Queues (p. 596) represent waiting lines. Insertions are made at the back (the tail; p. 596). Deletions are made from the front (the head; p. 596).
- Binary trees facilitate high-speed searching and sorting of data, efficient duplicate elimination, representing file-system directories and compiling expressions into machine language.

Section 12.2 Self-Referential Structures

- A self-referential structure (p. 597) contains a pointer member that points to a structure of the same type.
- Self-referential structures can be linked together to form lists, queues, stacks and trees.
- A NULL pointer indicates the end of a data structure.

Section 12.3 Dynamic Memory Management

• Creating and maintaining dynamic data structures requires dynamic memory management (p. 598).

- Functions malloc and free, and operator sizeof, are essential to dynamic memory allocation.
- Function malloc (p. 598) receives the number of bytes to be allocated and returns a void * pointer to the allocated memory. A void * pointer may be assigned to a variable of any pointer type.
- If no memory is available, malloc returns NULL.
- Function free (p. 598) deallocates memory so that it can be reallocated in the future.
- C also provides functions calloc and realloc for creating and modifying dynamic arrays.

Section 12.4 Linked Lists

- A linked list is a linear collection of self-referential structures, called nodes (p. 599), connected by pointer links (p. 599).
- A linked list is accessed via a pointer to the first node. Subsequent nodes are accessed via the link pointer member stored in each node.
- By convention, the link pointer in the last node of a list is set to NULL to mark the end of the list.
- Data is stored in a linked list dynamically—each node is created as necessary.
- A node can contain data of any type, including other struct objects.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- Linked-list nodes are not guaranteed to be stored contiguously in memory. Logically, however, the nodes of a linked list appear to be contiguous.

Section 12.5 Stacks

- A stack (p. 608) can be implemented as a constrained version of a linked list. New nodes are added to and removed from a stack only at the top.
- A stack is a last-in, first-out (LIFO; p. 609) data structure.
- The primary functions used to manipulate a stack are push and pop. Function push creates a new node and places it on top of the stack. Function pop removes a node from the top of the stack, frees the memory that was allocated to the popped node and returns the popped value.
- When you call a function, it must know how to return to its caller, so the caller's return address is pushed onto a stack. If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order so that each function can return to its caller. Stacks support recursive function calls in the same manner as conventional nonrecursive calls.
- Stacks are used by compilers in the process of evaluating expressions and generating machine-language code.

Section 12.6 Queues

- Queue nodes are removed only from the head of the queue and inserted only at the tail of the queue—referred to as a first-in, first-out (FIFO; p. 614) data structure.
- The insert and remove operations for a queue are known as enqueue and dequeue (p. 615).

Section 12.7 Trees

- A tree (p. 621) is a nonlinear, two-dimensional data structure. Tree nodes contain two or more links.
- Binary trees (p. 621) are trees whose nodes all contain two links.

- The root node (p. 621) is the first node in a tree. Each link in the root node of a binary tree refers to a child (p. 621). The left child (p. 621) is the first node in the left subtree (p. 621), and the right child (p. 621) is the first node in the right subtree (p. 621). A node's children are called siblings (p. 621).
- A node with no children is called a leaf node (p. 621).
- A binary search tree (with no duplicate node values; p. 621) has the characteristic that the values in any left subtree are less than the value in its parent node (p. 622), and the values in any right subtree are greater than the value in its parent node.
- A node can be inserted only as a leaf node in a binary search tree.
- The steps for an in-order traversal are: Traverse the left subtree in-order, process the value in the node, then traverse the right subtree in-order. The value in a node is not processed until the values in its left subtree are processed.
- The in-order traversal (p. 622) of a binary search tree processes the node values in ascending order. Creating a binary search tree actually sorts the data, so this process is called the binary tree sort (p. 625).
- The steps for a pre-order traversal (p. 622) are: Process the value in the node, traverse the left subtree pre-order, then traverse the right subtree pre-order. After this traversal processes a given node's value, it processes the node's left subtree values, then its right subtree values.
- The steps for a post-order traversal (p. 622) are: Traverse the left subtree post-order, traverse the right subtree post-order, then process the value in the node. The value in each node is not processed until the values of its children are processed.
- A binary search tree facilitates duplicate elimination (p. 626). As the tree is created, an attempt to insert a duplicate value will be recognized because a duplicate will follow the same "go left" or "go right" decisions on each comparison as the original value did. Thus, the duplicate will eventually be compared with a node in the tree containing the same value. The duplicate value may simply be discarded at this point.
- Searching a binary tree for a value that matches a key is fast. In a tightly packed tree, each level contains about twice as many elements as the previous level. A binary search tree with n elements would have a maximum of $\log_2 n$ levels, and thus it requires a maximum of $\log_2 n$ comparisons to find a match or to determine that no match exists. Searching a tightly packed 1,000-element binary search tree requires no more than 10 comparisons because $2^{10} > 1,000$. Searching a tightly packed 1,000,000-element binary search tree requires no more than 20 comparisons because $2^{20} > 1,000,000$.

Self-Review Exercises

12.1	Fill-In the blanks in each of the following:			
	a) A self structure is used to form dynamic data structures.			
	b) Function is used to dynamically allocate memory.			
	c) A(n) is a specialized version of a linked list in which nodes can be			
	inserted and deleted only from the start of the list.			
	d) Functions that look at a linked list but do not modify it are referred to as			
	·			
	e) A queue is referred to as a(n) data structure.			
	f) The pointer to the next node in a linked list is referred to as a(n)			
	g) Function is used to reclaim dynamically allocated memory.			

- h) A(n) _____ is a specialized version of a linked list in which nodes can be inserted only at the start of the list and deleted only from the end of the list.
- i) A(n) _____ is a nonlinear, two-dimensional data structure that contains nodes with two or more links.
- j) A stack is referred to as a(n) _____ data structure because the last node inserted is the first node removed.
- k) The nodes of a(n) _____ tree contain two link members.
- 1) The first node of a tree is the _____ node.
- m) Each link in a tree node points to a(n) _____ or ____ of that node.
- n) A tree node that has no children is called a(n) _____ node.
- o) The three traversal algorithms (covered in this chapter) for a binary tree are ______, _____ and _____.
- **12.2** (*Discussion*) What are the differences between a linked list and a stack?
- **12.3** (*Discussion*) What are the differences between a stack and a queue?
- **12.4** Write a statement or set of statements to accomplish each of the following. Assume that all the manipulations occur in main (therefore, no addresses of pointer variables are needed), and assume the following definitions:

```
struct gradeNode {
   char lastName[20];
   double grade;
   struct gradeNode *nextPtr;
};

typedef struct gradeNode GradeNode;
typedef GradeNode *GradeNodePtr;
```

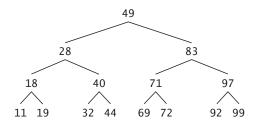
- a) Create a pointer to the start of the list called startPtr. The list is empty.
- b) Create a new node of type GradeNode that's pointed to by pointer newPtr of type GradeNodePtr. Assign the string "Jones" to member lastName and the value 91.5 to member grade (use strcpy). Provide any necessary declarations and statements.
- c) Assume that the list pointed to by startPtr currently consists of two nodes—one containing "Jones" and one containing "Smith". The nodes are in alphabetical order. Provide the statements necessary to insert nodes containing the following data for lastName and grade—be sure to insert the nodes in order:

"Adams" 85.0 "Thompson" 73.5 "Pritchard" 66.5

Use pointers previousPtr, currentPtr and newPtr to perform the insertions. State what previousPtr and currentPtr point to before each insertion. Assume newPtr always points to the new node and the new node has already been assigned the data.

d) Write a while loop that prints the data in each node of the list. Use pointer currentPtr to move along the list.

- e) Write a while loop that deletes all the nodes in the list and frees the memory associated with each node. Use pointer currentPtr and pointer tempPtr to walk along the list and free memory, respectively.
- **12.5** (*Binary Search Tree Traversals*) Provide the inorder, preorder and postorder traversals of the following binary search tree.



Answers to Self-Review Exercises

- **12.1** a) referential. b) malloc. c) stack. d) predicates. e) FIFO. f) link. g) free. h) queue. i) tree. j) LIFO. k) binary. l) root. m) child, subtree. n) leaf. o) inorder, preorder, postorder.
- **12.2** It's possible to insert a node anywhere in a linked list and remove a node from anywhere in a linked list. However, nodes in a stack may be inserted only at the top of the stack and removed only from the top of a stack.
- **12.3** A queue has pointers to both its head and its tail so that nodes may be inserted at the tail and deleted from the head. A stack has a single pointer to the top of the stack where both insertion and deletion of nodes is performed.
- **12.4** See the answers below:

```
a) GradeNodePtr startPtr = NULL;
b) GradeNodePtr newPtr;
  newPtr = malloc(sizeof(GradeNode));
  strcpy(newPtr->lastName, "Jones");
  newPtr->grade = 91.5;
  newPtr->nextPtr = NULL;
c) To insert "Adams":
  previousPtr is NULL, currentPtr points to the first element in the list.
  newPtr->nextPtr = currentPtr;
  startPtr = newPtr;
  To insert "Thompson":
  previousPtr points to the last element in the list (containing "Smith")
  currentPtr is NULL.
  newPtr->nextPtr = currentPtr;
  previousPtr->nextPtr = newPtr;
  To insert "Pritchard":
  previousPtr points to the node containing "Jones"
  currentPtr points to the node containing "Smith"
```

12.5 The *inorder* traversal is: 11 18 19 28 32 40 44 49 69 71 72 83 92 97 99 The *preorder* traversal is: 49 28 18 11 19 40 32 44 83 71 69 72 97 92 99 The *postorder* traversal is: 11 19 18 32 44 40 28 69 72 71 92 99 97 83 49

Exercises

- **12.6** (Concatenating Lists) Write a program that concatenates two linked lists of characters. The program should include function concatenate that takes pointers to both lists as arguments and concatenates the second list to the first list.
- **12.7** (*Merging Ordered Lists*) Write a program that merges two ordered lists of integers into a single ordered list. Function merge should receive pointers to the first node of each of the lists to be merged and return a pointer to the first node of the merged list.
- **12.8** (*Inserting into an Ordered List*) Write a program that inserts 25 random integers from 0 to 100 in order in a linked list. The program should calculate the elements' sum and floating-point average.
- **12.9** (*Creating a Linked List, Then Reversing Its Elements*) Write a program that creates a linked list of 10 characters, then creates a copy of the list in reverse order.
- **12.10** (*Reversing the Words of a Sentence*) Write a program that inputs a line of text and uses a stack to print the line reversed.
- **12.11** (*Palindrome Tester*) Write a program that uses a stack to determine whether a string is a palindrome (i.e., the string is spelled identically backward and forward). The program should ignore spaces and punctuation.
- **12.12** (Supermarket Simulation) Write a program that simulates a check-out line at a supermarket. The line is a queue. Customers arrive in random integer intervals of 1 to 4 minutes. Also, each customer is serviced in random integer intervals of 1 to 4 minutes. Obviously, the rates need to be balanced. If the average arrival rate is larger

than the average service rate, the queue will grow infinitely. Even with balanced rates, randomness can still cause long lines. Run the supermarket simulation for a 12-hour day (720 minutes) using the following algorithm:

- 1. Choose a random integer between 1 and 4 to determine the minute at which the first customer arrives.
- 2. At the first customer's arrival time:

Determine customer's service time (random int 1–4);

Begin servicing the customer;

Schedule arrival time of next customer (random int 1–4 added to current time).

3. For each minute of the day:

If the next customer arrives,

Say so;

Enqueue the customer;

Schedule the arrival time of the next customer.

If service was completed for the last customer,

Say so;

Dequeue next customer to be serviced;

Determine customer's service completion time

(random integer from 1 to 4 added to the current time).

Now run your simulation for 720 minutes and answer each of the following:

- a) What's the maximum number of customers in the queue at any time?
- b) What's the longest wait any one customer experiences?
- c) What happens if the arrival interval is changed from 1 to 4 minutes to 1 to 3 minutes?
- **12.13** (Allowing Duplicates in a Binary Tree) Modify the program of Fig. 12.4 to allow the binary tree to contain duplicate values.
- **12.14** (Binary Search Tree of Strings) Write a program based on the program of Fig. 12.4 that inputs a line of text, tokenizes the sentence into separate words, inserts the words in a binary search tree, and prints the inorder, preorder, and postorder traversals of the tree.

Read the line of text into an array. Use strtok to tokenize the text. When a token is found, create a new node for the tree, assign the pointer returned by strtok to member string of the new node, and insert the node in the tree.

12.15 (*Duplicate Elimination*) We've seen that duplicate elimination is straightforward when creating a binary search tree. Describe how you would perform duplicate elimination using only a one-dimensional array. Compare the performance of array-based duplicate elimination with the performance of binary-search-tree-based duplicate elimination.

- **12.16** (*Depth of a Binary Tree*) Write a function depth that receives a binary tree and determines how many levels it has.
- **12.17** (*Recursively Print a List Backward*) Write a function printListBackward that recursively outputs the items in a list in reverse order. Use your function in a test program that creates a sorted list of integers and prints the list in reverse order.
- **12.18** (*Recursively Search a List*) Write a function searchList that recursively searches a linked list for a specified value. The function should return a pointer to the value if it's found; otherwise, NULL should be returned. Use your function in a test program that creates a list of integers. The program should prompt the user for a value to locate in the list.
- **12.19** (Binary Tree Search) Write function binaryTreeSearch that attempts to locate a specified value in a binary search tree. The function should take as arguments a pointer to the root node of the binary tree and a search key to be located. If the node containing the search key is found, the function should return a pointer to that node; otherwise, the function should return a NULL pointer.
- **12.20** (*Level-Order Binary Tree Traversal*) The program of Fig. 12.4 illustrated three recursive methods of traversing a binary tree—inorder traversal, preorder traversal, and postorder traversal. This exercise presents the level-order traversal of a binary tree. This traversal processes the node values level-by-level from left-to-right starting at the root-node level. The level-order traversal is not a recursive algorithm. It uses the queue data structure to process the nodes in the correct order. The algorithm is as follows:
 - 1. Insert the root node in the queue.
 - 2. While there are nodes left in the queue,

Get the next node in the queue.

Print the node's value.

If the pointer to the left child of the node is not NULL

Insert the left child node in the queue.

If the pointer to the right child of the node is not NULL Insert the right child node in the queue.

Write function levelorder to perform a level order traversal of a binary tree. The function should take as an argument a pointer to the binary tree's root node. Modify the program of Fig. 12.4 to use this function. Compare the output from this function to the other traversals' outputs to see that it worked correctly. You'll need to modify and incorporate the queue-processing functions of Fig. 12.3 in this program.

12.21 (*Printing Trees*) Write a recursive function outputTree to display a binary tree. The function should output the tree row-by-row with the top of the tree at the left of the screen and the bottom of the tree toward the right of the screen. Each row is output vertically. For example, the binary tree in Exercise 12.5 is output as follows:



Note that the rightmost leaf node appears at the top of the output in the rightmost column, and the root node appears at the left of the output. Each column of output starts five spaces to the right of the previous column. Function outputTree should receive as arguments a pointer to the tree's root node and an integer totalSpaces representing the number of spaces preceding the value to output. This variable should start at zero so that the root node is output at the left of the screen. The function uses a modified inorder traversal to output the tree. The algorithm is as follows:

While the pointer to the current node is not NULL,

Recursively call outputTree with the current node's right subtree and totalSpaces + 5.

Use a for statement to count from 1 to totalSpaces and output spaces. Output the value in the current node.

Recursively call outputTree with the current node's left subtree and totalSpaces + 5.

Special Section: Systems Software Case Study—Building Your Own Compiler

In Exercise 7.28, we introduced the made-up Simpletron Machine Language (SML). In Exercise 7.29, you used simulation to create the Simpletron computer—a *virtual machine*—to execute programs written in SML. In this challenge section, you'll build a compiler that converts programs written in **Simple**—a made-up, concise, high-level programming language—to SML. This section "ties" together key pieces of the programming process. You'll:

- write several Simple high-level language programs,
- compile the programs using the compiler you'll build, generating SML machine-language code into a file,

- load the SML machine-language code from that file into the Simpletron's memory, and
- **execute** the SML machine-language programs on the Simpletron virtual machine you built in Exercise 7.29.

This section consists of six exercises. The first two cover some key computer-science technology you'll need to implement your compiler. The third introduces the Simple high-level language with some completely coded examples and asks you to write several of your own Simple programs. The fourth guides you through building your compiler. The fifth introduces the crucial topic of compiler optimization—you'll modify your compiler to reduce the number of SML instructions it generates, which will make your SML programs more memory efficient and enable them to execute faster. The final exercise gives you an opportunity to modify your compiler to add more useful features.

12.22 (*Infix-to-Postfix Converter*) Compilers use stacks to help evaluate expressions and generate machine-language code. In this and the next exercise, we investigate how compilers evaluate arithmetic expressions consisting only of single-digit integer constants, operators and parentheses. You can easily modify the algorithms we present to work with multiple-digit integers and floating-point numbers as well.

People generally write expressions like 3 + 4 and 7 / 9 with the operator *between* its operands. This is called **infix notation**. Computers "prefer" **postfix notation**, in which the operator is written *to the right* of its two operands. The preceding infix expressions would appear in postfix notation as 3 4 + and 7 9 /.

To evaluate an infix expression, some compilers

- first convert the expression to postfix notation, then
- evaluate the postfix version.

Each of these **stack-oriented algorithms** requires one left-to-right pass of the expression. In this exercise, you'll implement the **infix-to-postfix conversion algorithm**. In the next, you'll implement the **postfix-expression evaluation algorithm**.

Write a program that converts a valid infix arithmetic expression with singledigit integers such as

$$(6 + 2) * 5 - 8 / 4$$

to a postfix expression. The postfix version of the preceding infix expression is

Note that postfix expressions contain no parentheses. The program should read the expression into character array infix and use the stack functions implemented in this chapter to help create the postfix expression in character array postfix. The algorithm for creating a postfix expression is as follows:

- 1. Push a left parenthesis '(' onto the stack.
- 2. Append a right parenthesis ')' to the end of infix.
- 3. While the stack is not empty, read infix from left to right and do the following: If infix's current character is a digit, copy it to the next element of postfix. If infix's current character is a left parenthesis, push it onto the stack.

If infix's current character is an operator,

Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and insert the popped operators in postfix.

Push the current character in infix onto the stack.

If infix's current character is a right parenthesis,

Pop operators from the top of the stack and insert them in postfix until a left parenthesis is at the top of the stack.

Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:

- + addition
- subtraction
- multiplication
- / division

The stack should be maintained with the following declarations:

```
struct stackNode {
   char data;
   struct stackNode *nextPtr;
};

typedef struct stackNode StackNode;
typedef StackNode *StackNodePtr;
```

The program should consist of main and eight other functions with the following function prototypes:

Char pop(StackNodePtr *topPtr); Pop a value off the stack and return that value. Char stackTop(StackNodePtr topPtr); Return the stack's top value without popping the stack. bool isEmpty(StackNodePtr topPtr); Return true if the stack is empty (that is, topPtr is NULL); otherwise, return false. void printStack(StackNodePtr topPtr); Print the stack—this function traverses the linked list that implements the stack, but does not modify it.

12.23 (*Postfix-Expression Evaluator*) Write a program that evaluates a valid postfix expression such as

```
6 2 + 5 * 8 4 / -
```

The program should read a postfix expression consisting of single digits and operators into a character array. Postfix expressions do not contain parentheses—they're eliminated during the infix-to-postfix conversion. The program should scan the postfix expression and evaluate it using the following algorithm and the stack functions implemented earlier in this chapter:

- 1. Append the null character ('\0') to the end of the postfix expression. When the algorithm encounters this null character, the evaluation of the postfix expression is complete.
- 2. While the null character ('\0') has not been encountered, read the expression from left to right:

If the current character is a digit,

Push its integer value onto the stack. The integer value of a digit character is its value in the computer's character set minus the value of the zero character ('0') in the computer's character set.

Otherwise, if the current character is an operator,

Pop the stack's two top elements into variables x and y.

Calculate y operator x.

Push the calculation's result onto the stack.

3. When the null character ('\0') is encountered in the expression, pop the stack's top value. This is the postfix expression's result.

This algorithm supports only binary arithmetic operators. So in *Step 2*, if the operator is '/', the top of the stack is 2, and the next element in the stack is 8, you'd pop 2 into x, pop 8 into y, evaluate 8 / 2, and push the result, 4, back onto the stack. This applies to each binary arithmetic operator.

The arithmetic operations allowed in an expression are:

- addition
- subtraction
- * multiplication
- / division

The stack should be maintained with the following declarations:

```
struct stackNode {
   int data;
   struct stackNode *nextPtr;
};

typedef struct stackNode StackNode;
typedef StackNode *StackNodePtr;
```

The program should consist of main and six other functions with the following function prototypes:

```
Function prototype
                       Description
int evaluatePostfixExpression(char *expr);
                       Evaluate the postfix expression and return its result.
int calculate(int op1, int op2, char operator);
                       Evaluate the expression op1 operator op2 and return its result.
void push(StackNodePtr *topPtr, int value);
                       Push a value onto the stack.
int pop(StackNodePtr *topPtr);
                       Pop a value off the stack and return that value.
bool isEmpty(StackNodePtr topPtr);
                       Return true if the stack is empty (that is, topPtr is NULL); otherwise,
                       return false.
void printStack(StackNodePtr topPtr);
                       Print the stack—this function traverses the linked list that implements
                       the stack, but does not modify it.
```

12.24 (The Simple Programming Language—Writing Simple Programs) Before building the compiler, let's discuss a simple yet powerful, high-level language similar to early versions of the BASIC programming language. We call the language Simple. Every Simple statement consists of a line number and a Simple instruction. Line numbers must appear in ascending order. Each instruction begins with one of the following Simple commands: rem, input, let, print, goto, if...goto or end, which we describe in the following table. Simple evaluates only integer expressions using the +, -, * and / operators. These operators have the same precedence as in C. Parentheses can change an expression's order of evaluation. Exercise 12.27 suggests enhance-

ments to the Simple compiler. Several of the suggested enhancements, such as adding floating-point capability, require modifications to the Simpletron virtual machine as well.

Command	Example statement	Description
rem	50 rem this is a remark	Text following the command rem is for documentation purposes only and is ignored—no SML code is generated.
input	30 input x	Display a question mark to prompt the user to enter a single integer. Read that integer from the keyboard and store the integer in x.
let	80 let u = 4 * (j - 7))	Assign to u the value of 4 * (j - 7). An arbitrarily complex infix expression can appear to the right of the equal sign.
print	10 print w	Display the value of single integer variable w.
goto	70 goto 45	Transfer program control to line 45.
ifgoto	35 if i == z goto 80	Compare i and z for equality and transfer control to line 80 if the condition is true; otherwise, continue execution with the next statement.
end	99 end	Terminate program execution.

Additional Simple Language Rules

Simple also has the following language rules:

- The Simple compiler recognizes only lowercase letters—all characters in a Simple program should be lowercase.
- A variable name is a single letter. Multi-character variable names are not allowed, so Simple programs should document their variables in rem statements.
- Simple uses only int variables.
- Simple does not have variable declarations—merely mentioning a variable name in a program declares the variable and initializes it to zero.
- Simple's syntax **does not allow string manipulation** (reading a string, writing a string, comparing strings, etc.).
- Simple uses the **conditional branch if...goto statement** and the **unconditional branch goto statement** to alter a program's flow of control. If the condition in the if...goto statement is true, control transfers to the specified line number. The following relational and equality operators <, >, <=, >=, == or != are valid in an if...goto statement. Their precedence is the same as in C.

Our compiler assumes that Simple programs are entered correctly. Exercise 12.27 asks you to modify the compiler to perform **syntax-error checking**.

Simple Program Examples

Let's consider several Simple programs that demonstrate the language's features. The first (Fig. 12.5) reads two integers from the keyboard, stores the values in variables a and b, and computes and prints their sum (stored in variable c).

```
10 rem
         input two integers, then determine and print their sum
15 rem
20 rem
         input the two integers
30 input a
40 input b
45 rem
         add integers and store result in c
50 rem
60 let c = a + b
65 rem
70 rem
         print the result
80 print c
90 rem
         terminate program execution
99 end
```

Fig. 12.5 Input two integers, then determine and print their sum.

The next program (Fig. 12.6) determines and prints the larger of two integers. The integers are input from the keyboard and stored in the variables s and t. The if...goto statement tests the condition s >= t. If the condition is true, control transfers to line 90, which displays s. Otherwise, the program displays t, then transfers control to the end statement in line 99, where the program terminates.

```
input two integers, then determine and print the larger one
10 rem
20 input s
30 input t
32 rem
       test if s is greater than or equal to t
35 rem
40 if s >= t goto 90
45 rem
50 rem
        t is greater than s, so print t
60 print t
70 goto 99
75 rem
80 rem
         s is greater than or equal to t, so print s
90 print s
99 end
```

Fig. 12.6 Input two integers, then determine and print the larger one.

Simple does not have repetition statements like C's for, while or do...while. However, you can simulate each of these using the if...goto and goto statements. Figure 12.7 uses a sentinel-controlled loop to calculate the squares of several integers. Each integer is input from the keyboard and stored in variable j. If the value entered is the sentinel -9999, control transfers to line 99, where the program termi-

nates. Otherwise, k is assigned the square of j, k is output to the screen and control is passed to line 20, where the next integer is input.

```
10 rem
         calculate squares of integers until user enters -9999 sentinel to end
20 input j
23 rem
         test for sentinel value
25 rem
30 if j == -9999 goto 99
         calculate square of j and assign result to k
35 rem
40 let k = j * j
50 print k
53 rem
55 rem
         loop to get next j
60 goto 20
99 end
```

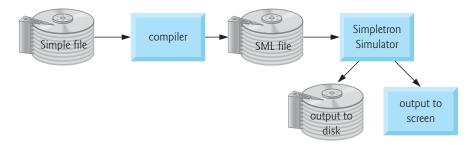
Fig. 12.7 | Calculate squares of integers until user enters -9999 to end.

Write Your Own Simple Programs

Using the sample programs of Figs. 12.5–12.7 as your guide, write Simple programs to accomplish each of the following:

- a) Input three integers, determine their average and print the result.
- b) Use a sentinel-controlled loop to input 10 integers and compute and print their sum.
- c) Use a counter-controlled loop to input seven integers, some positive and some negative, and compute and print their average.
- d) Input a series of integers and determine and print the largest. The first integer input indicates how many numbers should be processed.
- e) Input 10 integers and print the smallest.
- f) Calculate and print the sum of the even integers from 2 to 30.
- g) Calculate and print the product of the odd integers from 1 to 9.

12.25 (Building a Compiler; Prerequisite: Complete Exercises 7.28, 7.29, 12.22, 12.23 and 12.24) Now that we've presented the Simple language (Exercise 12.24), let's discuss how to build a Simple compiler. The following diagram summarizes the process for compiling a Simple program into SML, then executing it in the Simpletron simulator:



The compiler reads a file containing a Simple program, compiles it into SML code, then writes the SML one instruction per line to a text file. Next, the Simpletron simulator loads the SML file into the Simpletron's 100-element memory array, executes the program and outputs the results to the screen and to a file. We also send all screen outputs to a file to make it easy to print a hard copy.

The Simpletron simulator you developed in Exercise 7.29 takes its input from the keyboard, not a file. You must modify the Simpletron to read from a file so it can run the programs your Simple compiler produces.

The compiler performs **two passes** of a Simple program to convert it to SML:

- The first pass constructs a symbol table (discussed in detail below). The compiler stores in the symbol table every line number, variable name and constant of the Simple program. Each is stored with its type and its location in the final SML code. The first pass also produces the corresponding SML instruction(s) for each Simple statement. As you'll see, if the Simple program contains statements that transfer control to lines later in the program, the first pass results in an SML program containing some incomplete instructions.
- The second pass locates and completes the unfinished instructions and outputs the SML program to a file. The compiler's first pass code is much larger than its second pass code.

First Pass

The compiler begins by reading into memory the Simple program's first statement. The compiler separates the line into its individual tokens (i.e., "pieces" of a statement) for processing and compilation. You can use function strtok to do this. Recall that every statement begins with a line number followed by a command. As the compiler breaks the rest of the statement into tokens, if a token is a line number, a variable, or a constant, it's placed in the symbol table. A line number is placed in the symbol table only if it's the first token in a statement—you'll soon see what the compiler does with line numbers that are targets of conditional or unconditional branches.

The **symbolTable** is an **array** of **tableEntry** structures representing each **symbol** in the program. There's no restriction on the number of symbols that can appear in the program. So, the symbolTable could be large. Make the symbolTable a 200-element array for now. You can adjust its size once you have a working compiler.

The tableEntry structure is declared as follows:

```
struct tableEntry {
  int symbol;
  char type; // 'C' (constant), 'L' (line number), 'V' (variable)
  int location; // 00 to 99
};
```

Each tableEntry contains three members:

• **symbo1** is an integer containing a variable's ASCII representation (again, variable names are single characters), a line number, or an integer constant.

- type is a character indicating the symbol's type—'C' for a constant, 'L' for a line number, or 'V' for a variable.
- **location** contains the Simpletron memory location (00 to 99) associated with the symbol. Simpletron memory is an array of 100 integers in which SML instructions and data are stored. For a line number, the location is the Simpletron memory array element at which the Simple statement's SML instructions begin. For a variable or constant, the location is the Simpletron memory array element that stores the variable or constant. Variables and constants are allocated from location 99 of the Simpletron's memory downward. The first variable or constant is stored in location 99, the next in location 98, and so on.

The symbol table plays an integral part in converting Simple programs to SML. We learned in Exercise 7.28 that an SML instruction is a signed four-digit integer that comprises two parts—the operation code and the operand. The operation code is determined by the Simple command. For example, the Simple command input corresponds to SML operation code 10 (read), and the Simple command print corresponds to SML operation code 11 (write). The operand is a memory location containing the data on which the operation code performs its task. For instance, the operation code 10 reads a value from the keyboard and stores it in the memory location specified by the operand. The compiler searches symbol to determine the Simpletron memory location for each symbol so the corresponding location can be used to complete the SML instructions.

Each Simple statement's compilation process is based on the particular command. For example, after the line number in a rem statement is inserted in the symbol table, the compiler ignores the statement's remainder—a rem statement is for documentation purposes only—no SML code is generated. The input, print, goto and end statements correspond to the SML read, write, branch (to a specific location) and halt instructions. The compiler converts statements containing these Simple commands directly to SML. A goto statement may initially contain an unresolved reference if the specified line number refers to a statement later in the Simple program file. This is called a forward reference.

When a goto statement is compiled with an unresolved reference, the SML instruction must be flagged to indicate that the compiler's second pass must complete the instruction. The flags are stored in 100-element array int flags, in which each element is initialized to -1. If the memory location to which a line number refers is not yet known (that is, it's not in the symbol table), its line number is stored in array flags in the element with the same subscript as the incomplete instruction. The incomplete instruction's operand is set temporarily to 00. For example, an unconditional branch instruction (making a forward reference) is left as +4000 until the compiler's second pass, which we'll describe shortly.

Compiling an **if...goto** or **let statement** is more complicated than other statements—each produces more than one SML instruction. For an **if...goto statement**, the compiler produces code to test the condition and possibly branch to another line. The result of the branch could be an **unresolved forward reference**. Each Simple rela-

tional and equality operator can be simulated using SML's *branch zero* and *branch negative* instructions (or possibly a combination of both).

For a **let statement**, the compiler produces code to evaluate an **arbitrarily complex infix arithmetic expression** consisting of operators, single-letter integer variable names, integer constants and possibly parentheses. Expressions should separate each operand and operator with a space. Exercises 12.22–12.23 presented the **infix-to-post-fix conversion algorithm** and the **postfix-evaluation algorithm** compilers use to evaluate expressions. Before building your compiler, you should complete those exercises. The compiler converts each expression from **infix notation** to **postfix notation**, then evaluates the **postfix expression**. As you'll see, the compiler actually generates machine-language instructions in the process of performing the postfix expression evaluation.

How is it that the compiler produces the machine language to evaluate an expression containing variables? The postfix-evaluation algorithm contains a "hook" that allows our compiler to generate SML instructions rather than actually evaluating the expression. To enable this "hook" in the compiler, the postfix-evaluation algorithm must be modified to:

- search the **symbol table** for each **symbol** it encounters (and possibly insert it),
- determine the symbol's corresponding memory location, and
- push the memory location instead of the symbol onto the stack.

When an operator is encountered in the **postfix expression**, the stack's top two memory locations are popped, and SML for effecting the operation is produced using the **memory locations** as **operands**. Each subexpression's result is stored in a **temporary memory location** and **pushed back onto the stack**, so the postfix expression's evaluation can continue. When **postfix evaluation is complete**, the **result's memory location** is the **only location left on the stack**. This is popped, and SML instructions are generated to assign the result to the variable at the left of the let statement.

Second Pass

The compiler's second pass performs two tasks:

- resolve any unresolved references, and
- output the SML code to a file.

Resolution of each reference occurs as follows:

- 1. Search the flags array for an unresolved reference (i.e., an element with a value other than -1).
- 2. Locate the structure in array symbolTable containing the symbol stored in the flags array (be sure that the type of the symbol is 'L' for line number).
- 3. Insert the memory location from structure member location into the instruction with the unresolved reference (remember that an instruction containing an unresolved reference has operand 00).
- **4.** Repeat *Steps 1–3* until the end of the flags array is reached.

After the resolution process is complete, the compiler outputs the SML code to a file with one SML instruction per line. The Simpletron can read this file and execute its instructions (after the simulator is modified to read its input from a file, of course).

A Complete Example

The following example illustrates a complete conversion of a Simple program to SML. Consider a Simple program that inputs an integer, sums the values from 1 to that integer and prints that sum. So, if the user enters 4, the program would calculate 1 + 2 + 3 + 4, which is 10 and print that value. Figure 12.8 shows the program and the SML instructions produced by the compiler's first pass. Figure 12.9 shows the symbol table constructed by the compiler's first pass. Figure 12.10 shows how the compiler allocates Simpletron memory downward from cell 99. In a moment, we'll do a step-by-step walkthrough showing precisely how the compiler creates these tables.

Simple program	SML location and instruction	Description
5 rem sum 1 to x	none	rem ignored
10 input x	00 +1099	read x into location 99
15 rem check y == x	none	rem ignored
20 if $y == x$ goto 60	01 +2098	load y (location 98) into accumulator
	02 +3199	$\operatorname{sub} x$ (location 99) from accumulator
	03 +42 00	branch zero to unresolved location
25 rem increment y	none	rem ignored
30 let $y = y + 1$	04 +2098	load y (location 98) into accumulator
	05 +3097	add 1 (location 97) to accumulator
	06 +2196	store in temporary location 96
	07 +2096	load from temporary location 96
	08 +2198	store accumulator in y (location 98)
35 rem add y to total t	none	rem ignored
40 let $t = t + y$	09 +2095	load t (location 95) into accumulator
	10 +3098	add y (location 98) to accumulator
	11 +2194	store in temporary location 94
	12 +2094	load from temporary location 94
	13 +2195	store accumulator in t (location 95)
45 rem loop to $y == x$ test	none	rem ignored
50 goto 20	14 +4001	branch to location 01
55 rem output result	none	rem ignored
60 print t	15 +1195	output t (location 95) to screen
99 end	16 +4300	terminate execution
1		

Fig. 12.8 | SML instructions produced after the compiler's first pass.

Symbol	Type	Location			
5	L	00			
10	L	00			
'x'	V	99			
15	L	01			
20	L	01			
'y'	V	98			
25	L	04			
30	L	04			
1	С	97			
Temporary 96 allocated					
35	L	09			
40	L	09			
't'	V	95			
Temporary 94 allocated					
45	L	14			
50	L	14			
55	L	15			
60	L	15			
99	L	16			

Fig. 12.9 | Symbol table for program of Fig. 12.8.

Data counter	Value	Туре
93		Next Simpletron memory cell to allocate
94	none	Temporary variable
95	't'	Variable
96	none	Temporary variable
97	1	Constant
98	'y'	Variable
99	'x'	Variable

Fig. 12.10 | Compiler allocates Simpletron memory downward from the last cell of memory (99).

Most Simple statements convert directly to single SML instructions. The exceptions in this program are rem statements, the if...goto statement in line 20 and the 1et statements in lines 30 and 40. Remarks do not translate into machine language. However, the line number for a remark is placed in the symbol table in case the line number is referenced in a goto or an if...goto statement.

Line 20 of the program specifies that if the condition y == x is true, program control should transfer to line 60. Because line 60 appears *later* in the program, the compiler's **first pass** has not yet placed 60 in the symbol table (line numbers are placed in

the symbol table only when they appear as the first token in a statement the compiler has processed). Therefore, it's not possible at this time to determine the operand of the SML branch-zero instruction at location 03 in the array of SML instructions. The compiler places 60 in location 03 of the flags array to indicate that the second pass will complete this instruction.

We must keep track of the next instruction location in the SML array because there is not a one-to-one correspondence between Simple statements and SML instructions. For example, the if...goto statement of line 20 compiles into three SML instructions. Each time an instruction is produced, we must increment the instruction counter to the next SML array location. The Simpletron's limited memory size could present a problem for Simple programs with many statements, variables and constants. It's conceivable that the compiler could run out of Simpletron memory. To test for this case, your program should contain a data counter to keep track of the location at which the next variable or constant will be stored in the SML array. If the value of the instruction counter is larger than the data counter's value, the SML array is full. In this case, the compilation process should terminate, and the compiler should display an "out-of-memory" error message.

Step-by-Step Explanation of the Compilation Process's First Pass

Let's walk through the compilation process for the Simple program in Fig. 12.8. The compiler reads the first line of the program:

5 rem sum 1 to x

The first token in the statement (the line number) is determined using strtok (Chapter 8 discussed C's string-manipulation functions). The token returned by strtok is converted to an integer using atoi, so the symbol 5 can be placed in the symbol table. If the symbol is not found, it's inserted in the symbol table. Since we're at the beginning of the program and this is the first line, no symbols are in the table yet. So, 5 is inserted into the symbol table as type L (line number) and assigned the first location in the SML memory array (00). Although this line is a remark, a space in the symbol table is allocated for the line number (in case it's referenced by a goto or an if...goto). If a program branches to a rem statement's line number, control resumes with the first executable statement after the rem. No SML instruction is generated for a rem statement, so the instruction counter is not incremented.

Next, the compiler tokenizes the statement

10 input x

The line number 10 is placed in the symbol table as type L and assigned the first location in the SML array (00)—a remark began the program, so the instruction counter is still 00. The command input indicates that the next token is a variable (only a variable can appear as an argument in an input statement). Because input corresponds directly to an SML operation code, the compiler simply has to determine the location of x in the SML array. Symbol x is not found in the symbol table. So, it's inserted into the symbol table as the ASCII representation of x, given type V

(for variable), and assigned location 99 in the SML array. Data storage begins at 99 and is allocated downward—98, 97, and so on. SML code can now be generated for this statement. Operation code 10 (the SML *read* operation code) is multiplied by 100, and x's location (as determined in the symbol table) is added to it, which completes the instruction +1099. This is then stored in the SML array at location 00. The instruction counter is incremented by 1 because a single SML instruction was produced.

Next, the compiler tokenizes the statement

15 rem check
$$y == x$$

The symbol table is searched for line number 15, which is not found. The line number is inserted as type L and assigned the SML array's next location (01). Again, rem statements do not produce code, so the instruction counter is not incremented.

Next, the compiler tokenizes the statement

20 if
$$y == x$$
 goto 60

Line number 20 is inserted in the symbol table and given type L with the next location in the SML array, 01. The command if indicates that a condition is to be evaluated. The variable y is not found in the symbol table, so it's inserted and given the type V and the SML location 98. Next, SML instructions are generated to evaluate the condition. There is no direct equivalent in SML for the if...goto, so it must be simulated by performing a calculation using x and y and branching based on the result. If y equals x, the result of subtracting x from y is zero. So, the SML *branch-zero* instruction can be used with the calculation result to simulate the if...goto statement.

The first step requires that y be loaded (from SML location 98) into the accumulator. This produces the instruction 01 +2098. Next, x is subtracted from the accumulator. This produces the instruction 02 +3199. The value in the accumulator may be zero, positive or negative. Since the **operator is** ==, we want to *branch zero*. First, the symbol table is searched for the branch location (60), which is not found. So, 60 is placed in the flags array at location 03, and the instruction 03 +4200 is generated. We cannot add the branch location because we have not yet assigned a location to line 60 in the SML array—this location will be resolved later. The instruction counter is incremented to 04.

The compiler proceeds to the statement

25 rem increment y

The line number 25 is inserted in the symbol table as type L and assigned SML location 04. The instruction counter is not incremented.

When the statement

30 let
$$y = y + 1$$

is tokenized, the line number 30 is inserted in the symbol table as type L and assigned SML location 04. Command 1et indicates that the line is an assignment statement. First, all the symbols on the line are inserted in the symbol table (if they are not already there). The constant integer 1 is inserted as type C and assigned SML

location 97. Next, the right side of the assignment is **converted from infix to postfix notation**. Then the postfix expression (y 1 +) is evaluated. Symbol y is located in the symbol table, and its corresponding memory location, 98, is pushed onto the stack. Symbol 1 is also located in the symbol table, and its corresponding memory location, 97, is *pushed onto the stack*. When the operator + is encountered, the postfix evaluator *pops the stack* into the right operand of the operator and *pops the stack* again into the left operand of the operator, then produces the SML instructions

```
04 +2098 (load y)
05 +3097 (add 1)
```

The result of the expression is stored in a temporary location in memory (96) with instruction

```
06 +2196 (store temporary)
```

and the temporary location is *pushed onto the stack*. Now that the expression has been evaluated, the result must be stored in the let statement's variable y. So, the temporary location is loaded into the accumulator, and the accumulator is stored in y with the instructions

```
07 +2096 (load temporary)
08 +2198 (store y)
```

Notice that some of these SML instructions—storing the accumulator into temporary location 96, then immediately reloading the accumulator from location 96—appear to be **redundant**. Eliminating such redundancy is an example of **compiler optimization**, which we'll say more about shortly.

When the compiler tokenizes the statement

```
35 rem add y to total
```

it inserts line number 35 in the symbol table as type L and assigns it location 09.

The following statement is similar to line 30:

```
40 \text{ let } t = t + y
```

The variable t is inserted in the symbol table as type V and assigned SML location 95. The instructions follow the same logic and format as line 30, and the instructions 09 +2095, 10 +3098, 11 +2194, 12 +2094, and 13 +2195 are generated. The result of t + y is assigned to temporary location 94 before being assigned to t (95). The instructions in memory locations 11 and 12 also appear to be **redundant**. Again, we'll discuss this optimization issue shortly.

The statement

```
45 rem loop to y == x test
```

is a remark, so line 45 is inserted in the symbol table as type L and assigned SML location 14.

The statement

```
50 goto 20
```

transfers control to line 20. Line number 50 is inserted in the symbol table as type L and assigned SML location 14. The equivalent of goto in SML is the *unconditional branch* (40) instruction that transfers control to a specific SML location. The compiler searches the symbol table for line 20 and finds that it corresponds to SML location 01. The operation code (40) is multiplied by 100, and location 01 is added to produce the instruction +4001 at location 14.

The statement

55 rem output result

is a remark, so line 55 is inserted in the symbol table as type L and assigned SML location 15.

The statement

60 print t

is an output statement. Line number 60 is inserted in the symbol table as type L and assigned SML location 15. The equivalent of print in SML is operation code 11 (*write*). Variable t's location is determined from the symbol table, then added to the result of multiplying the operation code by 100. This forms the instruction +1195 at location 15.

The statement

99 end

is the final line of the program. Line number 99 is stored in the symbol table as type L and assigned SML location 16. The end command produces the SML instruction +4300 (43 is *halt* in SML). This is written as the final instruction in the SML memory array. The *halt* instruction has no operand. Can you think of a useful reason to allow an operand for the *halt* instruction?

The Compilation Process's Second Pass

On the compiler's *second pass*, we begin by searching the **flags array** for values other than -1. Location 03 contains 60, so the compiler knows that instruction 03 is incomplete. The compiler completes the instruction by searching the symbol table for 60, determining its location and adding the location to the incomplete instruction. In this case, the search determines that line 60 corresponds to SML location 15, so the completed instruction +4215 at location 03 is produced, replacing +4200. The Simple program has now been compiled successfully.

Building Your Compiler

To build the compiler, you'll have to perform each of the following tasks:

a) Modify the Simpletron simulator program you wrote in Exercise 7.29 to take its input from a file specified by the user (see Chapter 11). The simulator should also output its results to a file in the same format as the screen output.

- b) Modify the infix-to-postfix evaluation algorithm of Exercise 12.22 to process multi-digit integer operands and single-letter variable-name operands. Standard library function strtok can be used to locate each constant and variable in an expression. Constants can be converted from strings to integers using standard-library function atoi. The postfix expression's data representation must be altered to support variable names and integer constants.
- c) Modify the postfix evaluation algorithm to process multi-digit-integer operands and single-letter variable-name operands. The algorithm also should now implement the previously discussed "hook" so that it produces SML instructions rather than directly evaluating the expression. Standard-library function strtok can be used to locate each constant and variable in an expression, and constants can be converted from strings to integers using standard-library function atoi. The data representation of the postfix expression must be altered to support variable names and integer constants.
- d) Build the compiler—incorporate *Part b* and *Part c* for evaluating expressions in let statements. Your program should contain a function that performs the compiler's *first pass* and one that performs its *second pass*.

12.26 (Optimizing the Simple Compiler) When a program is compiled and converted into SML, a set of instructions is generated. Certain combinations of instructions often repeat themselves, usually in triplets called productions. A production normally consists of three instructions such as *load*, *add* and *store*. For example, Fig. 12.11 shows five of the SML instructions produced while compiling the program in Fig. 12.8. The first three instructions are the production that adds 1 to y. Instructions 06 and 07 store the accumulator value in temporary location 96, then load the value from that location right back into the accumulator so instruction 08 can store the value in location 98. Often a production is followed by a load instruction for the same location that was just stored. This code can be *optimized* by eliminating the *store* instruction and the subsequent *load* instruction that operate on the same memory location. This optimization would decrease the SML program's "memory footprint" by 25% and improve its execution speed. Figure 12.12 shows the **optimized SML** for the program of Fig. 12.8. There are *four fewer instructions in the optimized code*. Modify your compiler to perform the optimization you learned in this exercise.

```
04 +2098 (load)
05 +3097 (add)
06 +2196 (store)
07 +2096 (load)
08 +2198 (store)
```

Fig. 12.11 Unoptimized code from the program of Fig. 12.8.

Simple program	SML location and instruction	Description	
5 rem sum 1 to x	none	rem ignored	
10 input x	00 +1099	read x into location 99	
15 rem check y == x	none	rem ignored	
20 if $y == x$ goto 60	01 +2098	load y (98) into accumulator	
	02 +3199	sub x (99) from accumulator	
	03 +4211	branch to location 11 if zero	
25 rem increment y	none	rem ignored	
30 let $y = y + 1$	04 +2098	load y into accumulator	
	05 +3097	add 1 (97) to accumulator	
	06 +2198	store accumulator in y (98)	
35 rem add y to total	none	rem ignored	
40 let t = t + y	07 +2096	load t from location (96)	
	08 +3098	add y (98) to accumulator	
	09 +2196	store accumulator in t (96)	
45 rem loop to $y == x$ test	none	rem ignored	
50 goto 20	10 +4001	branch to location 01	
55 rem output result	none	rem ignored	
60 print t	11 +1196	output t (96) to screen	
99 end	12 +4300	terminate execution	

Fig. 12.12 Optimized code for the program of Fig. 12.8.

12.27 (Enhancing the Simple Compiler) Perform the following modifications to the Simple compiler. Some of these may also require modifications to the Simpletron Simulator program you wrote in Exercise 7.29. Many of these are quite challenging and could require substantial effort.

- a) Modify the Simpletron's memory to have 1000 cells (000–999). Modify the compiler to generate machine language appropriate for the 1000-element Simpletron memory.
- b) Allow the compiler to process floating-point values in addition to integers. The Simpletron Simulator must also be modified to process floating-point values.
- c) Add support for unary minus to specify negative integer values.
- d) Allow the **modulus operator** (%) to be used in 1et statements. Modify the Simpletron Machine Language to include a modulus instruction.
- e) Allow **exponentiation** in a let statement **using ^ as the exponentiation operator**. Modify the Simpletron Machine Language to include an exponentiation instruction.

- f) Allow the compiler to recognize uppercase and lowercase letters in Simple statements. So, x and X would be treated as different variables. No modifications to the Simpletron Simulator are required.
- g) Allow input statements to read values for multiple variables, such as input x, y. No modifications to the Simpletron Simulator are required.
- h) Allow the compiler to output multiple values in a single print statement, such as print a, b, c. This would output the variables' values, each separated from the next by one space. No modifications to the Simpletron Simulator are required.
- i) Allow the print statement's operand to be an **infix expression**.
- j) Add syntax-checking capabilities to the compiler so error messages are output when syntax errors are encountered in a Simple program. No modifications to the Simpletron Simulator are required.
- k) **Allow integer arrays**. No modifications to the Simpletron Simulator are required.
- 1) Allow subroutines specified by the Simple commands gosub and return. Command gosub passes program control to a subroutine, and command return passes control back to the statement after the gosub. This is similar to a function call in C. The same subroutine can be called from many gosubs distributed throughout a program. No modifications to the Simpletron Simulator are required.
- m) Allow repetition structures of the form

```
for x = 2 to 10
   rem Simple statements
next
```

This for statement loops from 2 to 10 with a default increment of 1. No modifications to the Simpletron Simulator are required.

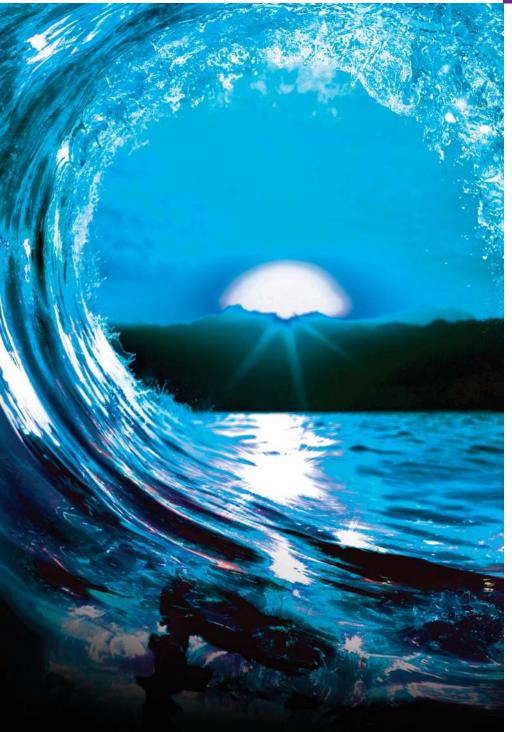
n) Allow repetition structures of the form

```
for x = 2 to 10 step 2
  rem Simple statements
next
```

This for statement loops from 2 to 10 with an increment of 2. The next line marks the end of the body of the for statement. No modifications to the Simpletron Simulator are required.

Computer-Science Thinking: Sorting Algorithms and Big O





Objectives

In this chapter, you'll:

- Sort an array using the selection sort algorithm.
- Sort an array using the insertion sort algorithm.
- Sort an array using the recursive merge sort algorithm.
- Learn about the efficiency of sorting algorithms and express it in "Big O" notation.
- Explore (in the exercises) other recursive sorts, including quicksort and a recursive selection sort.
- Explore (in the exercises) the high-performance bucket sort.

- 13.1 Introduction
- 13.2 Efficiency of Algorithms: Big O
 - 13.2.1 *O*(1) Algorithms
 - 13.2.2 O(n) Algorithms
 - 13.2.3 $O(n^2)$ Algorithms
- 13.3 Selection Sort
 - 13.3.1 Selection Sort Implementation
 - 13.3.2 Efficiency of Selection Sort

- **13.4** Insertion Sort
 - 13.4.1 Insertion Sort Implementation
 - 13.4.2 Efficiency of Insertion Sort
- **13.5** Case Study: Visualizing the High-Performance Merge Sort
 - 13.5.1 Merge Sort Implementation
 - 13.5.2 Efficiency of Merge Sort
 - 13.5.3 Summarizing Various Algorithms' Big O Notations

Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises

13.1 Introduction

In Chapter 6, you learned that sorting places data in ascending or descending order based on one or more sort keys. Here, we introduce the selection sort and insertion sort algorithms and the more efficient, but more complex, merge sort. We introduce **Big O notation**, which is used to estimate the worst-case run time for an algorithm—that is, how hard an algorithm may have to work to solve a problem.

For sorting arrays, it's important to understand that the result will be the same no matter which sorting algorithm you use. Your algorithm choice affects only the program's run time and memory use. The selection sort and insertion sort algorithms we study here are easy to program but inefficient. The third algorithm—recursive merge sort—is more efficient but harder to program.

The exercises present two more recursive sorts—quicksort and a recursive version of selection sort. Another exercise presents the bucket sort, which achieves high performance by cleverly using considerably more memory than the other sorts we discuss.

1

Self Check

I (Fill-In) Sorting places data in ascending or descending order based on one or more sort _____.

Answer: keys.

- **2** (*Multiple Choice*) Which of the following statements is *false*?
 - a) Big O notation estimates an algorithm's best-case run time—that is, how hard an algorithm may have to work to solve a problem.
 - b) In sorting, the sorted array will be the same no matter which sorting algorithm you use.
 - c) The sorting algorithm you choose affects your program's run time and memory use.
 - d) The selection sort and insertion sort algorithms are easy to program but inefficient. The recursive merge sort is more efficient but harder to program.

Answer: a) is false. Actually, Big O notation estimates the worst-case run time.

13.2 Efficiency of Algorithms: Big O



One way to describe an algorithm's effort is with Big O notation, which indicates how hard an algorithm may have to work to solve a problem. For searching and sorting algorithms, this depends mainly on how many data elements there are. In this chapter, we use Big O to describe various sorting algorithms' worst-case run times.

13.2.1 O(1) Algorithms

Suppose an algorithm tests whether an array's first element is equal to its second. If the array has 10 elements, this algorithm requires one comparison. If the array has 1,000 elements, the algorithm still requires one comparison. In fact, the algorithm is completely independent of the array's number of elements. This algorithm is said to have constant run time, which we represent in Big O notation as O(1) and pronounce "order 1." An O(1) algorithm does not necessarily require only one comparison. O(1) means that the number of comparisons is constant—it does not grow as the array size increases. An algorithm that tests whether the first array element is equal to any of the next three elements is still O(1), even though it requires three comparisons.

13.2.2 *O(n)* Algorithms

An algorithm that tests whether an array's first element is equal to *any* of the array's other elements requires at most n-1 comparisons, where n is the array's number of elements. If the array has 10 elements, this algorithm requires up to nine comparisons. If the array has 1,000 elements, this algorithm requires up to 999 comparisons.

As n grows larger, the n in the expression n-1 "dominates," so subtracting 1 becomes inconsequential. Big O is designed to highlight these dominant terms and ignore those that become unimportant as n grows. For this reason, an algorithm that requires n-1 comparisons is said to be O(n). An O(n) algorithm is referred to as having a linear run time. O(n) is often pronounced "on the order of n" or just "order n."

13.2.3 O(n²) Algorithms

Suppose an algorithm tests whether *any* array element is duplicated elsewhere in the array. The algorithm compares the first element with all of the array's other elements. The algorithm then compares the second element with all of the array's other elements except the first—the second was already compared to the first. Then, the algorithm compares the third element with all the other elements except the first two. In the end, this algorithm will end up making a total of (n-1) + (n-2) + ... + 2 + 1 or $n^2/2 - n/2$ comparisons. As n increases, the n^2 term dominates, and the n term becomes inconsequential. Big O notation highlights the n^2 term, leaving $n^2/2$. But as we'll soon see, constant factors are omitted in Big O notation.

Big O is concerned with how an algorithm's run time grows in relation to the number of items processed. Suppose an algorithm requires n^2 comparisons. With four elements, the algorithm will require 16 comparisons; with eight elements, the algorithm will require 64 comparisons. With this algorithm, doubling the number of

elements *quadruples* the number of comparisons. Consider a similar algorithm requiring $n^2/2$ comparisons. With four elements, the algorithm will require eight comparisons; with eight elements, the algorithm will require 32 comparisons. Again, doubling the number of elements *quadruples* the number of comparisons. Both algorithms grow as the square of n, so Big O ignores the constant and both algorithms are considered to be $O(n^2)$. This is referred to as *quadratic run time* and pronounced "on the order of n-squared" or simply "order n-squared."

When n is small, $O(n^2)$ algorithms (running on today's billion-operations-persecond personal computers) will not noticeably affect performance. But as n grows, you'll start to notice the performance degradation. An $O(n^2)$ algorithm running on a million-element array would require a trillion "operations," where each could actually require several machine instructions to execute. This could require a few hours to execute. A billion-element array would require a quintillion operations, a number so large that the algorithm could take decades! As you'll see in this chapter, $O(n^2)$ algorithms are easy to write. You'll also see an algorithm with a more favorable Big O measure. Efficient algorithms often require clever coding and more work to create. Their superior performance can be well worth the extra effort, especially as n gets large and as algorithms are combined into larger programs.

Self Check

I (*TruelFalse*) $O(n^2)$ algorithms running on today's billion-operations-per-second personal computers will not noticeably affect performance.

Answer: False. Actually, when n is small, $O(n^2)$ algorithms will not noticeably affect performance. But as n grows, you'll start to notice the performance degradation, even on today's powerful systems.

2 (*Fill-In*) Big O is concerned with how an algorithm's run time grows in relation to the _____.

Answer: number of items processed.

3 (True/False) An algorithm that is O(1) requires only one comparison.

Answer: False. O(1) means the number of comparisons is constant. The algorithm may require multiple comparisons, but that number does not grow as the array's size increases.

- **4** *(Fill-In)* An O(n) algorithm is referred to as having a _____ run time. **Answer:** linear.
- **5** (*Fill-In*) An $O(n^2)$ algorithm is referred to as having _____ run time. Answer: quadratic.

13.3 Selection Sort

Selection sort is a simple but inefficient sorting algorithm:

• The algorithm's first iteration selects the array's smallest element and swaps it with the array's first element.

- The second iteration selects the second-smallest element—which is the smallest of those remaining—and swaps it with the second element.
- The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last element. This leaves the largest element as the last.

After the *i*th iteration, the array's *i* smallest values will be sorted into increasing order in the array's first *i* elements.

As an example, consider the array

```
34 56 4 10 77 51 93 30 5 52
```

The selection sort first determines the array's smallest element (4), then swaps it with the value in element 0 (34), resulting in

```
4 56 34 10 77 51 93 30 5 52
```

The selection sort then determines the smallest remaining value beginning at element 1, which is the value 5 located in element 8. The program swaps 5 with the value 56 in element 1, resulting in

```
4 5 34 10 77 51 93 30 56 52
```

On the third iteration, the selection sort determines the next smallest value—10 in element 3—and swaps it with 34 in element 2, resulting in

```
4 5 10 34 77 51 93 30 56 52
```

The process continues until after nine iterations the array is fully sorted, as in

```
4 5 10 30 34 51 52 56 77 93
```

After the first iteration, the smallest element is in element 0. After the second iteration, the two smallest elements are in order in elements 0 and 1. After the third iteration, the three smallest elements are in order in elements 0–2, and so on.

13.3.1 Selection Sort Implementation

Figure 13.1 implements the selection sort algorithm. Lines 18–20 fill array with 10 random int values. The main function prints the unsorted array, passes array to the function selectionSort (line 29), then prints array again after it has been sorted.

```
// fig13_01.c
// The selection sort algorithm.

#define SIZE 10
#include <stdio.h>
#include <stdib.h>
#include <time.h>

// function prototypes
void selectionSort(int array[], size_t length);
void swap(int array[], size_t first, size_t second);
void printPass(int array[], size_t length, int pass, size_t index);
```

Fig. 13.1 The selection sort algorithm. (Part 1 of 3.)

```
12
    int main(void) {
13
       int array[SIZE] = {0}; // declare the array of ints to be sorted
14
15
16
       srand(time(NULL)); // seed the rand function
17
       for (size_t i = 0; i < SIZE; i++) {
18
19
          array[i] = rand() % 90 + 10; // give each element a value
20
21
22
       puts("Unsorted array:");
23
24
       for (size_t i = 0; i < SIZE; i++) { // print the array
          printf("%d ", array[i]);
25
26
27
28
       puts("\n");
29
       selectionSort(array, SIZE);
       puts("Sorted array:");
30
31
32
       for (size_t i = 0; i < SIZE; i++) { // print the array
33
          printf("%d ", array[i]);
34
       }
35
36
       puts("");
    }
37
38
39
    // function that selection sorts the array
    void selectionSort(int array[], size_t length) {
40
41
       // loop over length - 1 elements
       for (size_t i = 0; i < length - 1; i++) {
42
43
          size_t smallest = i; // first index of remaining array
44
45
          // loop to find index of smallest element
46
          for (size_t j = i + 1; j < length; j++) {
             if (array[j] < array[smallest]) {</pre>
47
48
                smallest = j;
49
             }
50
          }
51
52
          swap(array, i, smallest); // swap smallest element
53
          printPass(array, length, i + 1, smallest); // output pass
54
       }
55
    }
56
57
    // function that swaps two elements in the array
58
    void swap(int array[], size_t first, size_t second) {
       int temp = array[first];
59
60
       array[first] = array[second];
61
       array[second] = temp;
62
    }
63
```

Fig. 13.1 The selection sort algorithm. (Part 2 of 3.)

```
64
    // function that prints a pass of the algorithm
65
    void printPass(int array[], size_t length, int pass, size_t index) {
66
       printf("After pass %2d: ", pass);
67
68
       // output elements till selected item
69
       for (size_t i = 0; i < index; i++) {
          printf("%d ", array[i]);
70
71
72
73
       printf("%d* ", array[index]); // indicate swap
74
75
       // finish outputting array
       for (size_t i = index + 1; i < length; i++) {</pre>
76
          printf("%d ", array[i]);
77
78
79
80
       printf("%s", "\n
                                       "); // for alignment
81
82
       // indicate amount of array that is sorted
83
       for (int i = 0; i < pass; i++) {
          printf("%s", "-- ");
84
85
86
87
       puts(""); // add newline
88
    }
Unsorted array:
72 34 88 14 32 12 34 77 56
After pass
           1: 12
                   34
                        88
                            14
                                32
                                    72* 34
                                            77 56
                                                    83
After pass
            2: 12
                    14
                        88
                            34* 32
                                            77
                                                 56
                                                     83
                                    72
                                        34
After pass
            3: 12
                    14
                        32
                            34
                                88* 72
                                        34
                                             77
                                                 56
                                                     83
After pass
            4: 12
                        32
                           34* 88
                                    72
                                        34
                                             77
                                                 56
                                                     83
                    14
After pass
            5: 12
                   14
                       32
                           34
                                34
                                    72
                                        88* 77
                                                 56
                                                     83
After pass
            6: 12
                       32
                           34
                                34
                                    56
                                        88
                                            77
                                                 72* 83
                   14
After pass
            7: 12
                   14
                        32
                           34
                               34
                                    56
                                        72
                                            77
                                                 88* 83
After pass
            8: 12
                        32
                           34
                               34
                                    56
                                        72
                                            77* 88
                   14
                                                     83
After pass
           9: 12
                    14
                        32
                           34
                                34
                                    56
                                        72
                                             77
                                                 83
                                                     88*
After pass 10: 12
                                        72
                    14
                        32
                            34
                                34
                                    56
                                             77
                                                 83
                                                     88*
Sorted array:
```

Fig. 13.1 The selection sort algorithm. (Part 3 of 3.)

12 14 32 34 34 56 72 77 83 88

In function selectionSort (lines 40–55), variable smallest (line 43) stores the smallest remaining element's index. Lines 42–54 loop length - 1 times. Line 43 assigns to smallest the index i—the first index in the array's unsorted portion. Lines

46–50 process the remaining elements. For each, line 47 determines whether the element's value is less than the one at index smallest. If so, line 48 assigns the current element's index to smallest. After this loop, smallest contains the smallest remaining element's index. Line 52 calls swap (lines 58–62) to exchange the values at locations i and smallest, placing the smallest remaining element at position i in the array.

The output uses dashes to underline the portion of the array that's guaranteed to be sorted after each pass. We place an asterisk next to the element that was swapped with the smallest element on that pass. The element to the asterisk's left and the element above the rightmost dashes were the two values that were swapped on each pass.

13.3.2 Efficiency of Selection Sort

The selection sort algorithm runs in $O(n^2)$ time. In our selectionSort function, the outer loop (lines 42–54) processes the array's first n-1 elements, swapping the smallest remaining item into its sorted position. The inner loop (lines 46–50) processes the remaining items, searching for the smallest element. This loop executes n-1 times during the first outer-loop iteration, n-2 times during the second, then n-3, ..., 3, 2, 1. So, this inner loop iterates a total of n(n-1)/2 or $(n^2-n)/2$. In Big O notation, smaller terms drop out, and constants are ignored, leaving a Big O of $O(n^2)$.

Self Check

1 (*Discussion*) Consider the following array, which reflects the result of a selection sort's first pass:

```
4 56 34 10 77 51 93 30 5 52
```

What does the second pass do? Show the resulting array.

Answer: The second pass swaps 56 with 5 (the second smallest element), resulting in:

```
4 5 34 10 77 51 93 30 56 52
```

2 (*Code*) Consider the following selectionSort function:

```
void selectionSort(int array[], size_t length) {
2
       // loop over length - 1 elements
       for (size_t i = 0; i < length - 1; i++) {
3
4
          size_t smallest = i; // first index of remaining array
5
6
          // loop to find index of smallest element
7
          for (size_t j = i + 1; j < length; j++) {
             if (array[j] < array[smallest]) {</pre>
8
9
                 ???
10
          }
11
12
13
          swap(array, i, smallest); // swap smallest element
14
          printPass(array, length, i + 1, smallest); // output pass
15
       }
16
    }
```

What statement should replace the ??? in line 9 to complete the code?

Answer: smallest = j;

13.4 Insertion Sort

Insertion sort is another simple but inefficient sorting algorithm. This algorithm's first iteration takes the array's second element and, if it's less than the first element, swaps it with the first element. The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order. After this algorithm's *i*th iteration, the first *i* elements in the original array are sorted.

Consider as an example the following array:

34 56 4 10 77 51 93 30 5 52

The insertion sort's first iteration looks at the array's first two elements, 34 and 56. These elements are already in order, so the algorithm continues. If they were out of order, the algorithm would swap them.

In the next iteration, the algorithm looks at the third value, 4. This value is less than 56, so the algorithm stores 4 in a temporary variable and moves 56 one element to the right. The algorithm then determines that 4 is less than 34, so it moves 34 one element to the right. The program has now reached the beginning of the array, so it places 4 in element 0, resulting in

4 34 56 10 77 51 93 30 5 52

In the next iteration, the algorithm stores the value 10 in a temporary variable. Then the program compares 10 to 56 and moves 56 one element to the right because it's larger than 10. The program then compares 10 to 34, moving 34 right one element. When the program compares 10 to 4, it observes that 10 is larger than 4 and places 10 in element 1, resulting in

4 10 34 56 77 51 93 30 5 52

After the *i*th iteration, the array's first i + 1 elements are sorted with respect to one another. However, they may not be in their final locations, because there may be smaller values later in the array.

13.4.1 Insertion Sort Implementation

Figure 13.2 implements the insertion sort algorithm. In function insertionSort (lines 39–55), the variable insert (line 43) holds the element you're going to insert until we've moved the other elements. Lines 41–54 process the array's items from index 1 through the end. Each iteration stores in moveItem (line 42) the location where an item will be inserted and stores in insert (line 43) the value that will be inserted into its sorted portion. Lines 46–50 locate the position at which to insert the element. This loop terminates either when the algorithm reaches the front of the array or reaches an element that's less than the value to insert. Line 48 moves an element to the right, and line 49 decrements the position at which to insert the next element. After the nested loop ends, line 52 inserts the element into place. The program's output uses dashes to indicate the portion of the array that's sorted after each pass. We place an asterisk next to the element that was inserted into place on that pass.

```
// fig13_02.c
    // The insertion sort algorithm.
   #define SIZE 10
 3
 4 #include <stdio.h>
 5
   #include <stdlib.h>
    #include <time.h>
 6
 7
    // function prototypes
 8
 9
    void insertionSort(int array[], size_t length);
10
    void printPass(int array[], size_t length, int pass, size_t index);
11
12
    int main(void) {
13
       int array[SIZE] = \{0\}; // declare the array of ints to be sorted
14
15
       srand(time(NULL)); // seed the rand function
16
17
       for (size_t i = 0; i < SIZE; i++) {
18
          array[i] = rand() % 90 + 10; // give each element a value
19
       }
20
21
       puts("Unsorted array:");
22
       for (size_t i = 0; i < SIZE; i++) { // print the array
23
24
          printf("%d ", array[i]);
25
       }
26
27
       puts("\n");
28
       insertionSort(array, SIZE);
29
       puts("Sorted array:");
30
       for (size_t i = 0; i < SIZE; i++) { // print the array
31
32
          printf("%d ", array[i]);
33
       }
34
35
       puts("");
    }
36
37
38
    // function that sorts the array
    void insertionSort(int array[], size_t length) {
39
       // loop over length - 1 elements
40
41
       for (size_t i = 1; i < length; i++) {
           size_t moveItem = i; // initialize location to place element
42
           int insert = array[i]; // holds element to insert
43
44
45
           // search for place to put current element
46
           while (moveItem > 0 && array[moveItem - 1] > insert) {
47
               // shift element right one slot
               array[moveItem] = array[moveItem - 1];
48
49
               --moveItem;
           }
50
51
52
           array[moveItem] = insert; // place inserted element
```

Fig. 13.2 The insertion sort algorithm. (Part 1 of 2.)

```
54
       }
55
    }
56
57
    // function that prints a pass of the algorithm
58
    void printPass(int array[], size_t length, int pass, size_t index) {
       printf("After pass %2d: ", pass);
59
60
61
       // output elements till selected item
62
       for (size_t i = 0; i < index; i++) {
           printf("%d ", array[i]);
63
64
       }
65
66
       printf("%d* ", array[index]); // indicate swap
67
68
       // finish outputting array
69
       for (size_t i = index + 1; i < length; i++) {</pre>
           printf("%d ", array[i]);
70
       }
71
72
73
       printf("%s", "\n
                                       "); // for alignment
74
75
       // indicate amount of array that is sorted
76
       for (size_t i = 0; i <= pass; i++) {
           printf("%s", "-- ");
77
78
       }
79
80
       puts(""); // add newline
81
   }
Unsorted array:
72 16 11 92 63 99 59 82
                                99
                                     30
After pass
           1: 16* 72
                      11 92
                                63
                                    99
                                         59
                                             82
                                                 99
                                                     30
            2: 11* 16
                       72
                            92
                                    99
                                        59
                                             82
                                                 99
                                                     30
After pass
                                63
               ___
After pass
            3: 11
                    16
                        72
                            92* 63
                                    99
                                         59
                                             82
                                                 99
                                                     30
                        63* 72
                                92
                                             82
After pass
            4: 11 16
                                    99
                                         59
                                                 99
                                                     30
            5: 11
After pass
                   16
                        63
                            72
                                92
                                    99* 59
                                             82
                                                 99
                                                     30
            6: 11
                        59* 63
                                72
                                    92
                                             82
                                                     30
After pass
                    16
                                         99
                                                 99
                    --
                        --
                                --
After pass
            7: 11
                   16
                        59
                            63
                                72
                                    82* 92
                                             99
                                                 99
                                                     30
                            ___
                        59
                            63
                                72
                                    82
                                         92
                                             99
                                                 99* 30
After pass
            8: 11
                   16
After pass 9: 11
                   16
                        30* 59
                                63
                                    72
                                         82
                                             92
                                                 99
                                                     99
Sorted array:
11 16 30 59 63 72 82 92 99 99
```

printPass(array, length, i, moveItem);

Fig. 13.2 | The insertion sort algorithm. (Part 2 of 2.)

13.4.2 Efficiency of Insertion Sort

Like selection sort, the insertion sort algorithm runs in $O(n^2)$ time. Like our function selectionSort in Section 13.3.1, the insertionSort function uses nested loops. The outer loop (lines 41–54) iterates SIZE - 1 times, inserting an element into the appropriate position in the elements sorted so far. For this application's purposes, SIZE - 1 is equivalent to n-1, as SIZE is the array's number of elements. The nested loop (lines 46–50) iterates over the array's preceding elements. In the worst case, this while loop requires n-1 comparisons. Each individual loop runs in O(n) time. In Big O notation, you must multiply the number of iterations of each loop in nested loops. For each outer-loop iteration, there will be a certain number of inner-loop iterations. In this algorithm, for each O(n) outer-loop iterations, there will be O(n) inner-loop iterations. Multiplying these values results in a Big O of $O(n^2)$.

1

Self Check

- I (*Fill-In*) The insertion sort algorithm's first iteration takes the array's second element and, if it's less than the first element, swaps it with the first element. The second iteration looks at the third element and inserts it into the correct position with respect to the first two, so all three elements are in order. After the algorithm's *i*th iteration, the array's _____ elements will be sorted.

 Answer: first *i*.
- **2** (*Discussion*) The insertion sort and selection sort algorithms both run in $O(n^2)$ time. What program structure that they each have causes the $O(n^2)$ run time? **Answer:** They each have a nested for-loop.

13.5 Case Study: Visualizing the High-Performance Merge Sort

The merge sort algorithm is efficient but conceptually more complex than selection sort and insertion sort. The merge sort algorithm sorts an array by splitting it into two equal-sized subarrays, sorting each subarray, then merging them into one larger array. With an odd number of elements, the algorithm creates the two subarrays, such that one has one more element than the other.

Our merge sort implementation in this example is recursive. The base case is a one-element array, which is, of course, sorted. So, merge sort immediately returns when it's called with a one-element array. The recursion step splits an array of two or more elements into two equal-sized subarrays, recursively sorts each subarray, then merges them into one larger, sorted array. Again, if there are an odd number of elements, one subarray is one element larger than the other.

Suppose the algorithm has already merged smaller arrays to create sorted arrays A:

4 10 34 56 77 and B:

5 30 51 52 93

Merge sort combines these two arrays into one larger, sorted array. The smallest element in A is 4 (located in element 0). The smallest element in B is 5 (located in element 0). To determine the smallest element in the merged array, the algorithm compares 4 and 5. The value from A is smaller, so 4 becomes the first element in the merged array. Next, the algorithm compares 10 (element 1 in A) to 5 (element 0 in B). The value from B is smaller, so 5 becomes the second element in the merged array. The algorithm continues by comparing 10 to 30, with 10 becoming the third element in the array, and so on.

13.5.1 Merge Sort Implementation

Figure 13.3 implements the merge sort algorithm. Lines 35–37 define the mergeSort function. Line 36 calls function sortSubArray with the arguments 0 and 1ength - 1 (1ength is the array's size). The arguments are the beginning and ending subscripts of the array to sort, so this call to sortSubArray operates on the entire array. Lines 40–62 define the sortSubArray function. Line 42 tests the base case. If the subarray size is 1, the subarray is sorted, so the function simply returns immediately. If the subarray's size is greater than 1, the function splits the subarray in two, recursively calls function sortSubArray to sort the two halves, then merges them. Line 56 recursively calls function sortSubArray for the subarray's first half, and line 57 recursively calls function sortSubArray for the subarray's second half. When these two calls return, each half is sorted. Line 60 calls function merge (lines 65–111) on the two halves to combine them into one larger sorted subarray.

```
I // fig13_03.c
 2 // The merge sort algorithm.
 3 #define SIZE 10
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <time.h>
 8 // function prototypes
 9 void mergeSort(int array[], size_t length);
void sortSubArray(int array[], size_t low, size_t high);
   void merge(int array[], size_t left, size_t middle1,
П
       size_t middle2, size_t right);
12
    void displayElements(int array[], size_t length);
    void displaySubArray(int array[], size_t left, size_t right);
15
    int main(void) {
16
       int array[SIZE] = {0}; // declare the array of ints to be sorted
17
18
       srand(time(NULL)); // seed the rand function
19
20
21
       for (size_t i = 0; i < SIZE; i++) {
22
          array[i] = rand() % 90 + 10; // give each element a value
23
       }
24
```

Fig. 13.3 The merge sort algorithm. (Part 1 of 5.)

```
25
       puts("Unsorted array:");
26
       displayElements(array, SIZE); // print the array
27
       puts("\n");
       mergeSort(array, SIZE); // merge sort the array
28
29
       puts("Sorted array:");
30
       displayElements(array, SIZE); // print the array
31
       puts("");
32
    }
33
34
    // function that merge sorts the array
   void mergeSort(int array[], size_t length) {
36
       sortSubArray(array, 0, length - 1);
37
    }
38
39
   // function that sorts a piece of the array
   void sortSubArray(int array[], size_t low, size_t high) {
40
41
       // test base case: size of array is 1
42
       if ((high - low) >= 1) \{ // if not base case...
43
          size_t middle1 = (low + high) / 2;
44
          size_t middle2 = middle1 + 1;
45
46
          // output split step
          printf("%s", "split:
                                 ");
47
48
          displaySubArray(array, low, high);
          printf("%s", "\n
49
50
          displaySubArray(array, low, middle1);
                                 ");
          printf("%s", "\n
51
52
          displaySubArray(array, middle2, high);
53
          puts("\n");
54
55
          // split array in half and sort each half recursively
56
          sortSubArray(array, low, middle1); // first half
          sortSubArray(array, middle2, high); // second half
57
58
59
          // merge the two sorted arrays
60
          merge(array, low, middle1, middle2, high);
61
       }
62
    }
63
    // merge two sorted subarrays into one sorted subarray
65
   void merge(int array[], size_t left, size_t middle1,
66
       size_t middle2, size_t right) {
       size_t leftIndex = left; // index into left subarray
67
68
       size_t rightIndex = middle2; // index into right subarray
69
       size_t combinedIndex = left; // index into temporary array
70
       int tempArray[SIZE] = {0}; // temporary array
71
72
       // output two subarrays before merging
       printf("%s", "merge: ");
73
       displaySubArray(array, left, middle1);
74
75
       printf("%s", "\n
                              ");
76
       displaySubArray(array, middle2, right);
77
       puts("");
```

Fig. 13.3 | The merge sort algorithm. (Part 2 of 5.)

```
78
       // merge the subarrays until the end of one is reached
79
       while (leftIndex <= middle1 && rightIndex <= right) {</pre>
80
          // place the smaller of the two current elements in result
81
           // and move to the next space in the subarray
82
          if (array[leftIndex] <= array[rightIndex]) {</pre>
83
              tempArray[combinedIndex++] = array[leftIndex++];
84
           }
85
86
          else {
87
              tempArray[combinedIndex++] = array[rightIndex++];
88
       }
89
90
91
       if (leftIndex == middle2) { // if at end of left subarray ...
92
          while (rightIndex <= right) { // copy the right subarray</pre>
93
              tempArray[combinedIndex++] = array[rightIndex++];
94
           }
95
       }
       else { // if at end of right subarray...
96
97
          while (leftIndex <= middle1) { // copy the left subarray</pre>
98
              tempArray[combinedIndex++] = array[leftIndex++];
           }
99
       }
100
101
       // copy values back into original array
102
       for (size_t i = left; i <= right; i++) {</pre>
103
104
          array[i] = tempArray[i];
105
106
107
       // output merged subarray
       printf("%s", "
                                ");
108
109
       displaySubArray(array, left, right);
110
       puts("\n");
111 }
112
113 // display elements in array
II4 void displayElements(int array[], size_t length) {
115
       displaySubArray(array, 0, length - 1);
116 }
117
118 // display certain elements in array
II9 void displaySubArray(int array[], size_t left, size_t right) {
120
       // output spaces for alignment
121
       for (size_t i = 0; i < left; i++) {
          printf("%s", "
122
                            ");
123
       }
124
125
       // output elements left in array
126
       for (size_t i = left; i <= right; i++) {</pre>
127
          printf(" %d", array[i]);
128
       }
129 }
```

Fig. 13.3 The merge sort algorithm. (Part 3 of 5.)

```
Unsorted array:
 79 86 60 79 76 71 44 88 58 23
split:
          79 86 60 79 76 71 44 88 58 23
          79 86 60 79 76
                         71 44 88 58 23
split:
          79 86 60 79 76
          79 86 60
                   79 76
split:
          79 86 60
          79 86
                60
          79 86
split:
          79
             86
          79
merge:
             86
          79 86
merge:
          79 86
                60
          60 79 86
split:
                   79 76
                   79
                      76
merge:
                   79
                      76
                   76 79
merge:
          60 79 86
                   76 79
          60 76 79 79 86
split:
                          71 44 88 58 23
                          71 44 88
                                   58 23
split:
                          71 44 88
                          71 44
split:
                          71 44
                          71
                             44
                          71
merge:
                             44
                          44 71
                          44 71
merge:
                                88
                          44 71 88
```

Fig. 13.3 | The merge sort algorithm. (Part 4 of 5.)

```
split:
                                    58 23
                                    58
                                       23
                                    58
merge:
                                       23
                                    23 58
                          44 71 88
merge:
                                    23 58
                          23 44 58 71 88
          60 76 79 79 86
merge:
                          23 44 58 71 88
          23 44 58 60 71 76 79 79 86 88
Sorted array:
 23 44 58 60 71 76 79 79 86 88
```

Fig. 13.3 The merge sort algorithm. (Part 5 of 5.)

Lines 80–89 in function merge loop until reaching the end of either subarray. Line 83 tests which element at the beginning of the subarrays is smaller. If the left subarray element is smaller or equal, line 84 places it in position in the merged array. If the right subarray element is smaller, line 87 places it in position in the merged array. When the while loop completes, one entire subarray is placed in the merged array, but the other still contains data. Line 91 tests whether we reached the left subarray's end. If so, lines 92–94 add the right subarray's remaining elements to the merged array. Otherwise, we reached the right subarray's end, and lines 97–99 add the left subarray's remaining elements to the merged array. Finally, lines 103–105 copy tempArray's values into the correct portion of the original array. This program's output displays the splits and merges performed by merge sort, showing the sort's progress at each step of the algorithm.

13.5.2 Efficiency of Merge Sort

Merge sort is a far more efficient algorithm than either insertion sort or selection sort (although that may be difficult to believe when looking at the busy output in Fig. 13.3). Consider the first (nonrecursive) call to function sortSubArray. This results in

- two recursive calls to function sortSubArray with subarrays that are each approximately half the original array's size, and
- a single call to function merge.

The merge call requires, at worst, n-1 comparisons to fill the original array, which is O(n). Recall that each merged element is chosen by comparing one element from each subarray. The two calls to sortSubArray result in

- four more recursive calls to sortSubArray, each with a subarray approximately one-quarter the original array's size, and
- two more calls to function merge.

These two calls to the function merge each require, at worst, n/2-1 comparisons, for a total of O(n) comparisons. This process continues with each call to sortSubArray generating two additional calls to sortSubArray and a call to merge until the algorithm has split the original array into one-element subarrays. At each level, O(n) comparisons are required to merge the subarrays. Each level splits the arrays in half, so doubling the array size requires one more level. Quadrupling the array size requires two more levels. This pattern is logarithmic and results in $\log_2 n$ levels. This results in a total efficiency of $O(n \log n)$.

Supporting Exercises

This Merge Sort Visualization Case Study is supported by exercises on other complex sorts: Exercise 13.6 (Bucket Sort) and Exercise 13.7 (Quicksort). The bucket sort achieves high performance by cleverly using considerably more memory than the other sorts we discuss—this is an example of a space—time trade-off.

13.5.3 Summarizing Various Algorithms' Big O Notations

The following table summarizes the Big O of the sorting algorithms we've covered and the quicksort algorithm, which you'll implement in Exercise 13.7.

Algorithm	Big O
Insertion sort	$O(n^2)$
Selection sort	$O(n^2)$
Merge sort	$O(n \log n)$
Bubble sort	$O(n^2)$
Quicksort	Worst case: $O(n^2)$
	Average case: $O(n \log n)$

The following table lists the Big O values we've covered in this chapter along with a number of values for n to highlight the differences in the growth rates. The table includes $O(\log n)$, which is the Big O for binary search you learned in Chapter 6.

n	Approximate decimal value	O(log n)	O(n)	O(n log n)	O(n ²)
2^{10} 2^{20} 2^{30}	1000 1,000,000 1,000,000,000	10 20 30	2^{10} 2^{20} 2^{30}	$10 \cdot 2^{10} 20 \cdot 2^{20} 30 \cdot 2^{30}$	2^{20} 2^{40} 2^{60}

Self Check

I (*Discussion*) The recursive merge sort algorithm sorts an array by splitting it into two equal-size subarrays, sorting each subarray, then merging them into one larger

array. The subarrays in merge sort are not sorted with sorts we've covered, such as the bubble sort, selection sort or insertion sort. Explain how the subarrays actually are sorted in merge sort.

Answer: If the array has an even number of elements, the merge sort splits the array into two equal-size halves. If the array has an odd number of elements, one "half" has one more element than the other "half."

Each half is then recursively split into two smaller halves. This halving process continues until each half contains only one element. This is the base case of the recursion, because a one-element array is sorted.

Next, those individual elements are merged into a two-element sorted subarray based on their values. So, in answer to the question, the sorting is actually trivial. As the algorithm's recursion unwinds, merge sort keeps merging smaller sorted subarrays to form larger sorted subarrays. Every merge of two sorted subarrays results in a larger sorted subarray that's about double the size of the ones being merged. The last merge results in a sorted version of the original array.

2 (Discussion) Here are the first three lines of our merge sort example's output:

```
split: 79 86 60 79 76 71 44 88 58 23 79 86 60 79 76 71 44 88 58 23 71 44 88 58 23
```

and here are the last three lines of the output:

```
merge: 60 76 79 79 86
23 44 58 71 88
23 44 58 60 71 76 79 79 86 88
```

Compare these outptus. How are your observations consistent with how the merge sort works?

Answer: 1. In the *final merge phase*, each subarray corresponds to one of the unsorted subarrays produced by the algorithm's *first split pass*. 2. Each subarray entering the final merge phase is *sorted*. 3. The 10-element *final merged array* contains the same elements as the original unsorted array that entered the first split pass. 4. The 10-element *final merged array* is, in fact, *sorted*. The recursive merge sort splits the original unsorted array into smaller and smaller pieces until they are down to single-element pieces, which it ultimately merges to form the smallest sorted pieces of the original array. It keeps merging those sorted pieces to form larger and larger sorted pieces until it finally merges the two—now sorted—halves of the original unsorted array to form the final sorted array.

Summary

Section 13.1 Introduction

• Sorting involves arranging data into order.

Section 13.2, Efficiency of Algorithms: Big O

• One way to describe an algorithm's efficiency is with **Big O notation** (*O*; p. 658), which indicates how hard an algorithm may have to work to solve a problem.

- For searching and sorting algorithms, Big O describes how an algorithm's **amount of effort** varies, depending on how many elements are in the data.
- An *O*(1) algorithm is said to have a **constant run time** (p. 659). This does not mean that the algorithm requires only one comparison. It just means that the number of comparisons does not grow as the size of the array increases.
- An O(n) algorithm is referred to as having a linear run time (p. 659).
- An $O(n^2)$ algorithm is referred to as having a quadratic run time (p. 660).
- Big O is designed to highlight dominant factors and ignore terms that become unimportant with high values of *n*.
- Big O notation is concerned with the growth rate of algorithm run times, so constants are ignored.

Section 13.3, Selection Sort

- The first iteration of a **selection sort** (p. 660) selects the smallest element in the array and swaps it with the first element. The second iteration selects the second-smallest element (the smallest of those remaining) and swaps it with the second element. Selection sort continues until the last iteration selects the second-largest element and swaps it with the second-to-last, leaving the largest element as the last. At the *i*th iteration of selection sort, the array's smallest *i* elements are sorted into the array's first *i* positions.
- The selection sort algorithm runs in $O(n^2)$ time (p. 664).

Section 13.4, Insertion Sort

- The first iteration of **insertion sort** (p. 665) takes the second element in the array and, if it's less than the first element, swaps it with the first element. The second iteration of insertion sort looks at the third element and inserts it in the correct position with respect to the first two elements. After the *i*th iteration of insertion sort, the first *i* elements in the original array are sorted. Only n-1 iterations are required.
- The insertion sort algorithm runs in $O(n^2)$ time (p. 668).

Section 13.5, Case Study: Visualizing the High-Performance Merge Sort

- The merge sort algorithm (p. 668) is faster but more complex to implement than selection sort and insertion sort.
- The merge sort algorithm sorts an array by **splitting** it into two equal-size **subarrays**, sorting each subarray and **merging** the subarrays into one larger array.
- Merge sort's base case is an array with one element, which is already sorted, so merge sort
 immediately returns when it's called with a one-element array. The merge part of merge sort
 takes two sorted arrays (these could be one-element arrays) and combines them into one
 larger sorted array.
- The merge is performed by looking at each array's first element, which is also the smallest element. Merge sort places the smallest of these in the first element of the larger, sorted array. If there are still elements in the subarray, merge sort looks at the second element in that subarray (which is now the smallest element remaining) and compares it to the first element in the other subarray. Merge sort continues this process until all the elements in one of the subarrays has been processed. Then, merge sort adds the remaining elements of the other subarray to the larger array.
- The merging portion of the merge sort algorithm is performed on two subarrays, each of approximately size n/2. Creating each subarray requires n/2-1 comparisons for each subar-

- ray, or O(n) comparisons total. This pattern continues, as each level works on twice as many arrays, but each is half the previous array's size.
- This halving results in log n levels, each level requiring O(n) comparisons, for a total efficiency of $O(n \log n)$ (p. 674), which is far more efficient than $O(n^2)$.

Self-Review Exercises

- **13.1** Fill-In the blanks in each of the following statements:
 - a) A selection sort application would take approximately _____ times as long to run on a 128-element array as on a 32-element array.
 - b) The efficiency of merge sort is _____.
- **13.2** The Big O of the linear search is O(n), and the Big O of the binary search is $O(\log n)$. What key aspect of both the binary search (Chapter 6) and the merge sort accounts for the logarithmic portion of their respective Big Os?
- **13.3** In what sense is the insertion sort superior to the merge sort? In what sense is the merge sort superior to the insertion sort?
- **13.4** In the text, we say that after the merge sort splits the array into two subarrays, it then sorts these two subarrays and merges them. Why might someone be puzzled by our statement that "it then sorts these two subarrays"?

Answers to Self-Review Exercises

- **13.1** a) 16, because an $O(n^2)$ algorithm takes 16 times as long to sort four times as much information. b) $O(n \log n)$.
- **13.2** Both algorithms incorporate "halving"—somehow reducing something by half on each pass. The binary search eliminates from consideration one-half of the array after each comparison. The merge sort splits the array in half each time it's called.
- **13.3** The insertion sort is easier to understand and implement than the merge sort. The merge sort is far more efficient— $O(n \log n)$ —than the insertion sort— $O(n^2)$.
- **13.4** In a sense, it does not really sort these two subarrays. It simply keeps splitting the original array in half until it provides a one-element subarray, which is, of course, sorted. It then builds up the original two subarrays by merging these one-element arrays to form larger subarrays, which are then merged, and so on.

Exercises

13.5 (*Recursive Selection Sort*) A selection sort searches an array looking for the array's smallest element. When that element is found, it's swapped with the first element of the array. The process is then repeated for the subarray, beginning with the second element. Each pass of the array results in one element being placed in its proper location. This sort requires processing capabilities similar to bubble sort—for an array of n elements, n-1 passes must be made, and for each subarray, n-1 comparisons must be made to find the smallest value. When the subarray being processed

contains one element, the array is sorted. Write a recursive function selectionSort to perform this algorithm.

13.6 (*Bucket Sort*) A bucket sort begins with a one-dimensional array of positive integers to sort, and a two-dimensional array of integers with rows subscripted from 0 to 9 and columns subscripted from 0 to n-1, where n is the array's number of values to sort. Each row of the two-dimensional array is a "bucket." In this exercise, you'll write a bucketSort function that takes as arguments an int array and its size.

The algorithm is as follows:

- a) Loop through the one-dimensional array and, based on each value's ones digit, place the value in a bucket (a row of the two-dimensional bucket array). For example, place 97 in row 7, 3 in row 3 and 100 in row 0.
- b) Loop through the bucket array's rows and columns and copy the values back to the original array. The new order of the above values in the one-dimensional array is 100, 3 and 97.
- c) Repeat this process for each subsequent digit position (tens, hundreds, thousands, and so on) and stop when the largest number's leftmost digit has been processed.

The second pass of the array places 100 in row 0, 3 in row 0 (it had only one digit, so we treat it as 03) and 97 in row 9. After this pass, the values' order in the one-dimensional array is 100, 3 and 97. The third pass places 100 in row 1, 3 (003) in row zero and 97 (097) in row zero (after 3). The bucket sort is guaranteed to properly sort all the values after processing the leftmost digit of the largest number. The bucket sort knows it's done when all the values are copied into row zero of the two-dimensional bucket array.

The two-dimensional bucket array is ten times the size of the int array being sorted. This sorting technique provides far better performance than, for example, a bubble sort but requires much larger storage capacity. Bubble sort requires only one additional memory location for the type of data being sorted. Bucket sort is an example of a space—time trade-off. It uses more memory but performs better.

The bucket sort algorithm described above requires copying all the data back to the original array on each pass. Another possibility is to create a second two-dimensional bucket array and repeatedly move the data between the two bucket arrays until all the data is copied into row zero of one of the arrays. Row zero then contains the sorted array.

- **13.7** (*Quicksort*) We discussed various sorting techniques in the examples and exercises of Chapter 6 and this chapter. We now present the recursive quicksort sorting technique. The basic algorithm for a one-dimensional array of values is as follows:
 - a) Partitioning Step: Take the unsorted array's first element and determine its final location in the sorted array. That's the position for which all values to the element's left are less than that value, and all values to the element's right are greater than that value. We now have one element in its proper location and two unsorted subarrays.
 - b) Recursive Step: Perform the partitioning step on each unsorted subarray.

For each subarray, the *partitioning step* places another element in its final location of the sorted array and creates two more unsorted subarrays. A subarray consisting of one element is sorted, so that element is in its final location.

The basic algorithm seems simple enough, but how do we determine the final position of each subarray's first element? As an example, consider the following set of values—the element in bold is the partitioning element that will be placed in its final location in the sorted array:

37 2 6 4 89 8 10 12 68 45

a) Starting from the rightmost array element, compare each element with 37 until an element less than 37 is found, then swap 37 and that element. The first element less than 37 is 12, so we swap 37 and 12. In the updated array below, we show 12 in italic to indicate that it was just swapped with 37:

12 2 6 4 89 8 10 **37** 68 45

b) Starting from the array's left, but beginning with the element *after* 12, compare each element with 37 until an element greater than 37 is found, then swap 37 and that element. The first element greater than 37 is 89, so we swap 37 and 89. The updated array is

12 2 6 4 **37** 8 10 89 68 45

c) Starting from the right, but beginning with the element *before* 89, compare each element with 37 until an element less than 37 is found, then swap 37 and that element. The first element less than 37 is 10, so we swap 37 and 10. The updated array is

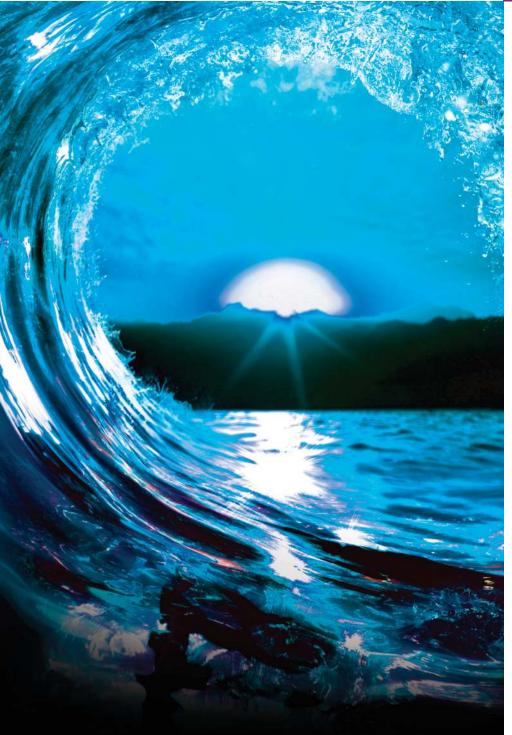
12 2 6 4 10 8 37 89 68 45

d) Starting from the left, but beginning with the element *after* 10, compare each element with 37 until an element greater than 37 is found, then swap 37 and that element. There are no more elements greater than 37. When we compare 37 with itself, we know that 37 is in its final location in the sorted array.

Once the partition has been applied to the array, there are two unsorted subarrays. The subarray with values less than 37 contains 12, 2, 6, 4, 10 and 8. The subarray with values greater than 37 contains 89, 68 and 45. Quicksort continues by partitioning both subarrays in the same manner as the original array.

Write recursive function quicksort to sort a one-dimensional integer array. The function should receive as arguments an int array, a starting subscript and an ending subscript. The quicksort should call the function partition to perform the partitioning step.

Preprocessor



14

Objectives

In this chapter, you'll:

- Use #include to help manage files in large programs.
- Use #define to create macros with and without arguments.
- Use conditional compilation to specify portions of a program that should not always be compiled, such as code that assists you in debugging.
- Display error messages during conditional compilation.
- Use assertions to test whether the expression values are correct.

- **14.1** Introduction
- 14.2 #include Preprocessor Directive
- **14.3 #define** Preprocessor Directive: Symbolic Constants
- **14.4** #define Preprocessor Directive: Macros
 - 14.4.1 Macro with One Argument
 - 14.4.2 Macro with Two Arguments
 - 14.4.3 Macro Continuation Character
 - 14.4.4 #undef Preprocessor Directive
 - 14.4.5 Standard-Library Macros
 - 14.4.6 Do Not Place Expressions with Side Effects in Macros

- **14.5** Conditional Compilation
 - 14.5.1 #if...#endif Preprocessor Directive
 - 14.5.2 Commenting Out Blocks of Code with #if...#endif
 - 14.5.3 Conditionally Compiling Debug Code
- **14.6** #error and #pragma Preprocessor Directives
- **14.7** # and ## Operators
- 14.8 Line Numbers
- **14.9** Predefined Symbolic Constants
- 14.10 Assertions
- 14.11 Secure C Programming

Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises

14.1 Introduction

The C preprocessor executes before each program compiles. It:

- includes other files into the file being compiled,
- · defines symbolic constants and macros,
- conditionally compiles program code and
- conditionally executes preprocessor directives.

Preprocessor directives begin with #. Only whitespace characters and comments delimited by /* and */ may appear before a preprocessor directive on a line.

C has perhaps the largest installed base of "legacy code" of any modern programming language. It's been in use for about five decades. As a professional C programmer, you're likely to encounter code written many years ago using older programming techniques. This chapter presents several of those techniques and recommends some newer techniques that can replace them.

1

Self Check

- I (Multiple Choice) Which of the following are actions performed by the preprocessor?
 - a) The inclusion of other files into the file being compiled.
 - b) Definition of symbolic constants and macros.
 - c) Conditional compilation of program code and conditional execution of preprocessor directives.
 - d) All of the above.

Answer: d.

2 (*True/False*) C has perhaps the largest installed base of "legacy code" of any modern programming language. It's been in active use for about five decades.

Answer: True.

14.2 #include Preprocessor Directive

You've used the #include preprocessor directive throughout this book. When the preprocessor encounters a #include, it replaces the directive with a copy of the specified file. The two forms of the #include directive are:

```
#include <filename>
#include "filename"
```

The difference between these is the location where the preprocessor begins searching for the file. For filenames enclosed in angle brackets (< and >)—such as standard library headers—the preprocessor searches in *implementation-dependent* compiler and system folders. You typically use filenames enclosed in quotes ("") to include headers that you define for use with your program. In this case, the preprocessor begins searching in the *same* folder as the file in which the #include directive appears. If the compiler cannot find the specified file in the current folder, it searches the implementation-dependent compiler and system folders.

In addition to using #include for standard library headers, you'll frequently use it in programs consisting of *multiple source files*. You'll often create headers for a program's common declarations, then include that file into multiple source files. Examples of such declarations are:

- struct and union declarations,
- typedefs,
- enums, and
- function prototypes.

Self Check

I (*Fill-In*) The _____ preprocessor directive causes a copy of a specified file to be included in place of the directive.

Answer: #include.

2 (*Fill-In*) If the filename in an #include directive is enclosed in quotes, the preprocessor begins its search for the file in _____. **Answer:** the same directory as the file being compiled.

14.3 #define Preprocessor Directive: Symbolic Constants

The **#define** directive creates:

- symbolic constants—constants represented as identifiers, and
- macros—operations defined as symbols.

The #define directive's format is

```
#define identifier replacement-text
```

By convention, a symbolic constant's *identifier* should contain only uppercase letters and underscores. Using meaningful names for symbolic constants helps make programs self-documenting.

Replacing Symbolic Constants

When the preprocessor encounters a #define directive, it replaces *identifier* with replacement text throughout that source file, ignoring any occurrences of identifier in string literals or comments. For example,

```
#define PI 3.14159
```

replaces all subsequent occurrences of the symbolic constant PI with 3.14159. Symbolic constants enable you to create named constants and use their names throughout the program.

Common Error with Symbolic Constants

Everything to the right of a symbolic constant's name replaces the symbolic constant. For example,

```
#define PI = 3.14159
```



causes the preprocessor to replace *every* occurrence of PI with "= 3.14159". Incorrect ERR (X) #define directives cause many subtle logic and syntax errors. So, in preference to the preceding #define, you may prefer to use const variables, such as

```
const double PI = 3.14159;
```

These have the additional benefit that they're defined in C, so the compiler can check them for proper syntax and type safety.

Self Check

- (Fill-In) The #define directive creates _____ constants and ____. **Answer:** symbolic, macros.
- **2** (*True/False*) The statement #define PI 3.14159;

replaces all subsequent occurrences of the symbolic constant PI with the numeric constant 3.14159.

Answer: False. Actually, this #define is a common error. Preprocessor directives are not C statements and generally should not end in semicolons. The above statement would replace all occurrences of PI with 3.14159; (including the semicolon), probably causing one or more compilation errors.

14.4 #define Preprocessor Directive: Macros

Technically, any identifier defined in a #define preprocessor directive is a macro. As with symbolic constants, the macro-identifier is replaced with replacement-text before the program is compiled. Macros may be defined with or without arguments. A macro without arguments is a symbolic constant. When the preprocessor encounters a macro with arguments, it substitutes the arguments in the replacement text, then **expands** the macro—that is, it replaces the macro with its replacement-text and argument list.

14.4.1 Macro with One Argument

Consider the following one-argument macro definition that calculates a circle's area:

```
#define CIRCLE_AREA(x) ((PI) * (x) * (x))
```

Expanding a Macro with an Argument

Wherever CIRCLE_AREA(*argument*) appears in the file, the preprocessor:

- substitutes *argument* for x in the replacement-text,
- replaces PI with its value 3.14159 (from Section 14.3), and
- expands the macro in the program.

For example, the preprocessor expands

```
double area = CIRCLE_AREA(4);
to
double area = ((3.14159) * (4) * (4));
```

At compile time, the compiler evaluates the preceding expression and assigns the result to the variable area.

Importance of Parentheses

The *parentheses* around each x in the replacement-text force the proper evaluation order when a macro's argument is an expression. Consider the statement

```
double area = CIRCLE_AREA(c + 2);
which expands to
    double area = ((3.14159) * (c + 2) * (c + 2));
```

This evaluates correctly because the parentheses force the proper evaluation order. If you omit the macro definition's parentheses, the macro expansion is

```
double area = 3.14159 * c + 2 * c + 2;
which evaluates incorrectly as
double area = (3.14159 * c) + (2 * c) + 2;
```



because of C's operator precedence rules. For this reason, you should always enclose macro arguments in parentheses in the replacement-text to prevent logic errors.

It's Better to Use a Function

Defining the CIRCLE_AREA macro as a function is safer. The circleArea function

```
double circleArea(double x) {
   return 3.14159 * x * x;
}
```

performs the same calculation as macro CIRCLE_AREA, but a function's argument is evaluated only once when the function is called. Also, the compiler performs type checking on functions. The preprocessor does not support type checking. In the past, programmers often used macros to replace function calls with inline code to eliminate the function-call overhead. Today's optimizing compilers often inline function calls for you, so many programmers no longer use macros for this purpose. You can also use the C standard's inline keyword (see Appendix C).

14.4.2 Macro with Two Arguments

The following two-argument macro calculates a rectangle's area:

```
#define RECTANGLE_AREA(x, y) ((x) * (y))
```

Wherever RECTANGLE_AREA(x, y) appears in the program, the preprocessor substitutes the values of x and y in the macro's replacement-text and expands the macro in the program. For example, the statement

```
int rectangleArea = RECTANGLE_AREA(a + 4, b + 7);
is expanded to
  int rectangleArea = ((a + 4) * (b + 7));
```

14.4.3 Macro Continuation Character

A macro's or symbolic constant's replacement-text is everything to the identifier's right in the #define directive. If the replacement-text is longer than the remainder of the line, you can place a backslash (\) continuation character at the end of the line to continue the replacement-text on the next line.

14.4.4 #undef Preprocessor Directive

Symbolic constants and macros can be discarded for the remainder of a source file using the #undef preprocessor directive. Directive #undef undefines a symbolic constant or macro name. A macro's or symbolic constant's scope is from its definition until it's undefined with #undef, or until the end of the source file. Once undefined, a macro or symbolic constant can be redefined with #define.

14.4.5 Standard-Library Macros

Some standard-library functions actually are defined as macros, based on other library functions. A macro commonly defined in the <stdio.h> header is

```
#define getchar() getc(stdin)
```

The macro definition of getchar uses function getc to get one character from the standard input stream. The <stdio.h> header's putchar function and the <ctype.h> header's character-handling functions often are implemented as macros as well.

14.4.6 Do Not Place Expressions with Side Effects in Macros

Expressions with *side effects* (e.g., variable values are modified) should *not* be passed to a macro, because macro arguments may be evaluated more than once. We'll show an example of this in Section 14.11.



Self Check

I (True/False) The following two-argument macro calculates a rectangle's area:

```
#define RECTANGLE_AREA(x, y) ((x) * (y))
```

Wherever RECTANGLE_AREA(x, y) appears in the program, the values of x and y are substituted in the macro replacement-text and the macro is expanded in place of the macro name. For example, the statement

```
double rectArea = RECTANGLE_AREA(a + 4, b + 7);
is expanded to
     double rectArea = (a + 4 * b + 7);
```

The value of the expression is evaluated at runtime and assigned to variable rectArea. Answer: False. Actually, the correct expansion is:

```
double rectArea = ((a + 4) * (b + 7));
```

2 (Fill-In) Expressions with ______ should not be passed to a macro because macro arguments may be evaluated more than once. **Answer:** side effects.

14.5 Conditional Compilation

Conditional compilation enables you to control which preprocessor directives execute and whether parts of your C code compile. Each conditional preprocessor directive evaluates a constant integer expression. Cast expressions, sizeof expressions and enumeration constants *cannot* be evaluated in preprocessor directives.

14.5.1 #if...#endif Preprocessor Directive

The conditional preprocessor construct is much like the if selection statement. Consider the following preprocessor code:

```
#if !defined(MY CONSTANT)
   #define MY_CONSTANT 0
#endif
```

This determines whether MY_CONSTANT is defined—that is, whether MY_CONSTANT has already appeared in an earlier #define directive within the current source file. The expression defined (MY_CONSTANT) evaluates to 1 (true) if MY_CONSTANT is defined; otherwise, it evaluates to 0 (false). If the result is 0, !defined(MY_CONSTANT) evaluates to 1, indicating that MY_CONSTANT was not defined previously, so the #define directive executes. Otherwise, the preprocessor skips the #define directive.

Every #if construct ends with #endif. The directives #ifdef and #ifndef are shorthand for #if defined(name) and #if!defined(name). You can test a multiple-part conditional preprocessor construct by using

- #elif (the equivalent of else if in an if statement) and
- #else (the equivalent of else in an if statement) directives.

Conditional preprocessor directives are frequently used to *prevent header files from being included multiple times in the same source file*. These directives frequently are used to enable and disable code that makes software compatible with a range of platforms.

14.5.2 Commenting Out Blocks of Code with #if...#endif

During program development, it's often helpful to "comment out" portions of your code to prevent them from being compiled. If the code contains multiline comments, /* and */ cannot do this because you cannot nest multiline comments. Instead, you can use the following preprocessor construct:

```
#if 0
    code prevented from compiling
#endif
```

To enable the code to be compiled, replace the 0 in the preceding construct with 1.

14.5.3 Conditionally Compiling Debug Code

Conditional compilation is sometimes used as a *debugging* aid. For example, some programmers use printf statements to print variable values and to confirm a program's flow of control. You can enclose such printf statements in conditional preprocessor directives so the statements are compiled only while you're still debugging your code. For example,

```
#ifdef DEBUG
    printf("Variable x = %d\n", x);
#endif
```

compiles the printf statement if the symbolic constant DEBUG is defined with

```
#define DEBUG
```

before #ifdef DEBUG. When you complete your debugging phase, you remove or comment out the #define directive in the source file, and the printf statements inserted for debugging purposes are ignored during compilation. In larger programs, you might define several symbolic constants that control the conditional compilation in separate sections of the source file.

Many compilers allow you to define and undefine symbolic constants like DEBUG with a compiler flag that you supply each time you compile the code so that you do not need to change the code. When inserting conditionally compiled printf statements in locations where C expects a single statement (e.g., a control statement's body), ensure that the conditionally compiled statements are enclosed in braces ({}}).



I (True/False) The conditional compilation statement

```
#ifdef DEBUG
    printf("Variable x = %d\n", x);
#endif
```

compiles the printf statement if the symbolic constant DEBUG is defined (#define DEBUG) before #ifdef DEBUG.

Answer: True.

2 (*True/False*) During program development, it's often helpful to "comment out" portions of code to prevent them from being compiled. If the code contains multiline comments, /* and */ should be used to accomplish this task.

Answer: False. Actually, if the code contains multiline comments, /* and */ cannot be used to accomplish this task, because such comments cannot be nested. Instead, you can use the following preprocessor construct:

```
#if 0
    code prevented from compiling
#endif
```

14.6 #error and #pragma Preprocessor Directives

The #error directive

```
#error tokens
```

prints an implementation-dependent message, including the *tokens* specified in the directive. The tokens are sequences of characters separated by spaces. For example,

```
#error 1 - Out of range error
```

contains 6 tokens. When the #error directive is processed on some systems, the tokens are displayed as an error message, preprocessing stops, and the program does not compile.

The #pragma directive

```
#pragma tokens
```

causes an *implementation-defined* action. A #pragma not recognized by the implementation is ignored. For more information on #error and #pragma, see the documentation for your C compiler.

Self Check

I (*Fill-In*) When a(n) _____ preprocessor directive is processed on some systems, the tokens in the directive are displayed as an error message, preprocessing stops and the program does not compile.

Answer: #error.

2 (*Fill-In*) The #pragma directive causes a(n) _____ action Answer: implementation-defined.

14.7 # and ## Operators

The # operator converts a replacement-text token to a string surrounded by quotes. Consider the following macro definition:

```
#define HELLO(x) puts("Hello, " #x);
```

When HELLO(John) appears in a program file, the preprocessor expands it to

```
puts("Hello, " "John");
```

replacing #x with the string "John". Strings separated by whitespace are concatenated during preprocessing, so the preceding statement is equivalent to

```
puts("Hello, John");
```

The # operator must be used in a macro with arguments because #'s operand refers to one of the macro's arguments.

The ## operator *concatenates two tokens*. Consider the following macro definition:

```
#define TOKENCONCAT(x, y) x ## y
```

When TOKENCONCAT appears in a file, the preprocessor concatenates the arguments and uses the result to replace the macro. For example, TOKENCONCAT(0, K) is replaced by OK in the program. The ## operator must have two operands.

\checkmark

Self Check

1 (*Fill-In*) The _____ preprocessor operator causes a replacement-text token to be converted to a string surrounded by quotes.

Answer: #.

2 (*Code*) The ## preprocessor operator concatenates two tokens. Write the macro definition described by, "When SIDEBYSIDE appears in the program, its arguments are concatenated and used to replace the macro. So, SIDEBYSIDE(GOOD, BYE) is replaced by GOODBYE in the program."

Answer: #define SIDEBYSIDE(a, b) a ## b

14.8 Line Numbers

The #line preprocessor directive causes the subsequent source-code lines to be renumbered, starting with the specified constant integer value. The directive

```
#line 100
```

starts line numbering from 100 beginning with the next source-code line. Including a filename in the #line directive, as in

```
#line 100 "file1.c"
```

indicates that lines are numbered from 100 beginning with the next source-code line and that the filename for the purpose of any compiler messages is "file1.c". This version of the #line directive normally helps make the messages produced by syntax errors and compiler warnings more meaningful. The line numbers do not appear in the source file.

✓ Self Check

I (*True/False*) The preprocessor directive

```
#line 100 "file1.c"
```

indicates that lines are numbered from 100 beginning with the next source-code line and that the name of the file for the purpose of any compiler messages is "file1.c". Answer: *True*.

14.9 Predefined Symbolic Constants

Standard C provides predefined symbolic constants, several of which are shown in the following table:

Symbolic constant	Explanation
LINE	The line number of the current source-code line (an integer constant).
FILE	The name of the source file (a string).
DATE	The date the source file was compiled (in the form "Mmm dd yyyy", such as "Jan 19 2002").
TIME	The time the source file was compiled (in the form "hh:mm:ss").
STDC	The value 1 if the compiler supports Standard C; 0 otherwise. Requires the compiler flag /Za in Visual C++.

The remaining predefined symbolic constants are in Section 6.10.8 of the C standard. These identifiers begin and end with *two* underscores and often are useful for including additional information in error messages. These identifiers and the defined identifier (used in Section 14.5) cannot be used in #define or #undef directives.

Self Check

(Fill-In) The predefined symbolic constant ______ is described by, "The value 1 if the compiler supports Standard C; 0 otherwise."
 Answer: __STDC__.

14.10 Assertions

The assert macro—defined in <assert.h>—tests an expression's value at execution time. If the value is false (0), assert prints an error message and terminates the program by calling function abort of the general utilities library (<stdlib.h>).

The assert macro is a useful debugging tool for testing whether a variable has a correct value. For example, suppose variable x should never be larger than 10 in a program. You can use an assertion to test x's value and print an error message if it's greater than 10, as in

```
assert(x \ll 10);
```

If x is greater than 10 when this statement executes, the program displays an error message containing the line number and filename where the assert statement appears, then terminates. You'd then focus on this area of the code to find the error.

If the symbolic constant NDEBUG is defined, subsequent assertions in the source file are *ignored*. So, when assertions are no longer needed, rather than deleting each assertion manually, you can insert the following line in the source file:

```
#define NDEBUG
```

Many compilers have debug and release modes that automatically define and undefine NDEBUG.

Assertions are not meant as a substitute for error handling during normal runtime conditions. You should use them only to find logic errors during program development. The C standard also includes a capability called _Static_assert, which is essentially a compile-time version of assert that produces a *compilation error* if the assertion fails. We discuss _Static_assert in Appendix C.

Self Check

I (*True/False*) The assert macro—defined in <assert.h>—tests the value of an expression at compile time.

Answer: False. Actually, the assert macro tests the value of an expression at execution time. _Static_assert is essentially a compile-time version of assert that produces a compilation error if the assertion fails.

2 (*Fill-In*) When assertions are no longer needed, you can insert the line _____ in the code file rather than delete each assertion manually.

Answer: #define NDEBUG.

14.11 Secure C Programming

The CIRCLE_AREA macro defined in Section 14.4:

```
#define CIRCLE_AREA(x) ((PI) * (x) * (x))
```

is an *unsafe* macro, because it evaluates its argument x *more than once*. This can cause subtle errors. If the macro argument contains *side effects*—such as incrementing a variable or calling a function that modifies a variable's value—those side effects would be performed *multiple* times.

For example, if we call CIRCLE_AREA as follows:

```
double result = CIRCLE_AREA(++radius);
```

The preprocessor expands this to

```
double result = ((3.14159) * (++radius) * (++radius));
```

which increments radius *twice*. Also, the preceding statement's result is *undefined*, because C allows a variable to be modified *only once* in a statement. In a function call, the argument is *evaluated only once* before it's passed to the function. So, functions are always preferred to unsafe macros.



I (Fill-In) The CIRCLE_AREA macro

is considered unsafe because it _____. This can cause subtle errors.

Answer: evaluates its argument x more than once.

2 (*True/False*) Macros are always preferred to functions.

Answer: *False.* Actually, functions are always preferred to unsafe macros.

Summary

Section 14.1 Introduction

- The preprocessor executes before a program compiles.
- All preprocessor directives (p. 683) begin with #.
- Only whitespace characters and comments may appear before a preprocessor directive on a line.

Section 14.2 #include Presprocessor Directive

• The #include directive (p. 683) includes a copy of the specified file. If the filename is enclosed in quotes, the preprocessor begins searching in the same folder as the file being compiled. If the filename is enclosed in angle brackets (< and >), as is the case for C standard library headers, the search is performed in an implementation-defined manner.

Section 14.3 #define Preprocessor Directive: Symbolic Constants

- The #define preprocessor directive (p. 683) creates symbolic constants and macros.
- A symbolic constant (p. 683) is a name for a constant.

Section 14.4 #define Preprocessor Directive: Macros

- A macro is an operation defined in a #define preprocessor directive. Macros may be defined with or without arguments.
- Replacement-text (p. 684) is specified after a symbolic constant's identifier or after the closing right parenthesis of a macro's argument list. If the replacement-text for a macro or symbolic constant is longer than the remainder of the line, use a backslash (\; p. 686) at the end of the line to indicate that the replacement-text continues on the next line.
- Symbolic constants and macros can be discarded using the **#undef preprocessor directive** (p. 686). Directive **#undef** "undefines" the symbolic constant or macro name.
- The **scope** (p. 686) of a symbolic constant or macro is from its definition until it's undefined with #undef or until the end of the file.

Section 14.5 Conditional Compilation

- Conditional compilation (p. 688) enables you to control whether preprocessor directives execute and whether program code compiles.
- The conditional preprocessor directives evaluate constant integer expressions. Cast expressions, sizeof expressions and enum constants cannot be evaluated in preprocessor directives.
- Every #if construct ends with #endif (p. 688).

- Directives #ifdef and #ifndef (p. 688) are provided as shorthand for #if defined(name) and #if!defined(name).
- Multiple-part conditional preprocessor constructs may be tested with directives #elif and #else (p. 688).

Section 14.6 #error and #pragma Preprocessor Directives

- The #error directive (p. 689) terminates preprocessing, prevents compilation and prints an implementation-dependent message that includes the tokens specified in the directive.
- The **#pragma** directive (p. 689) causes an implementation-defined action. If the **#pragma** is not recognized by the implementation, it's ignored.

Section 14.7 # and ## Operators

- The # operator converts a *replacement-text* token to a string surrounded by quotes. The # operator must be used in a macro with arguments because #'s operand must be one of the macro's arguments.
- The **## operator** concatenates two tokens. The **##** operator must have two operands.

Section 14.8 Line Numbers

• The #1ine preprocessor directive (p. 690) causes the subsequent source-code lines to be renumbered, starting with the specified constant integer value. This directive also enables you to specify the filename used for that source-code file in compiler error messages.

Section 14.9 Predefined Symbolic Constants

- Constant __LINE__ (p. 691) is the line number (an integer) of the current source-code line.
- Constant __FILE__ (p. 691) is the name of the file (a string).
- Constant __DATE__ (p. 691) is the date the source file is compiled (a string).
- Constant __TIME__ (p. 691) is the time the source file is compiled (a string).
- Constant __STDC__ (p. 691) indicates whether the compiler supports Standard C.
- Each of the predefined symbolic constants begins and ends with two underscores.

Section 14.10 Assertions

• Macro assert (p. 691; <assert.h> header) tests the value of an expression. If the value is 0 (false), assert prints an error message and calls function abort (p. 691) to terminate program execution.

Self-Review Exercises

14.1	Fill-In the blanks in each of the following:
	a) Every preprocessor directive must begin with
	b) The conditional compilation construct may be extended to test for multiple
	cases by using the and directives.
	c) The directive creates macros and symbolic constants.
	d) Only characters may appear before a preprocessor directive on a line.
	e) The directive discards symbolic constant and macro names.
	f) The and directives are provided as shorthand notation
	for #if defined(name) and #if Idefined(name)

- g) _____ enables you to control whether preprocessor directives execute and whether code compiles.
- h) The _____ macro prints a message and terminates program execution if the macro's expression evaluates is 0.
- i) The _____ directive inserts a file in another file.
- j) The _____ preprocessor operator concatenates its two arguments.
- k) The _____ preprocessor operator converts its operand to a string.
- l) The character _____ indicates that the replacement-text for a symbolic constant or macro continues on the next line.
- m)The _____ directive causes the source-code lines to be numbered from the indicated value beginning with the next source-code line.
- **14.2** Write a program to print the values of the predefined symbolic constants listed in Section 14.9.
- **14.3** Write a preprocessor directive to accomplish each of the following:
 - a) Define the symbolic constant YES to have the value 1.
 - b) Define the symbolic constant NO to have the value 0.
 - c) Include the header common.h. The header is found in the same directory as the file being compiled.
 - d) Renumber the remaining lines in the file beginning with line number 3000.
 - e) If the symbolic constant TRUE is defined, undefine it and redefine it as 1. Do not use #ifdef.
 - f) If the symbolic constant TRUE is defined, undefine it and redefine it as 1. Use the #ifdef preprocessor directive.
 - g) If the symbolic constant TRUE is not equal to 0, define symbolic constant FALSE as 0. Otherwise define FALSE as 1.
 - h) Define the macro CUBE_VOLUME that computes the volume of a cube. The macro takes one argument.

Answers to Self-Review Exercises

- **14.1** a) #. b) #elif, #else. c) #define. d) whitespace. e) #undef. f) #ifdef, #ifndef. g) Conditional compilation. h) assert. i) #include. j) ##. k) #. l) \. m) #line.
- **14.2** See below. [*Note:* In Visual Studio, __STDC__ works requires the /Za compiler flag.]

```
1  // ex14_02.c
2  // Print the values of the predefined macros
3  #include <stdio.h>
4  int main(void) {
5     printf("_LINE_ = %d\n", _LINE__);
6     printf("_FILE_ = %s\n", _FILE__);
7     printf("_DATE_ = %s\n", _DATE__);
8     printf("_TIME_ = %s\n", _TIME__);
9     printf("_STDC_ = %d\n", _STDC__);
10 }
```

```
__LINE__ = 5

__FILE__ = ex14_02.c

__DATE__ = Jan 01 2021

__TIME__ = 11:39:12

__STDC__ = 1
```

14.3 See the answers below:

```
a) #define YES 1
b) #define NO 0
c) #include "common.h"
d) #line 3000
e) #if defined(TRUE)
     #undef TRUE
     #define TRUE 1
  #endif
f) #ifdef TRUE
     #undef TRUE
     #define TRUE 1
  #endif
g) #if TRUE
     #define FALSE 0
  #else
     #define FALSE 1
  #endif
h) #define CUBE_VOLUME(x) ((x) * (x) * (x))
```

Exercises

14.4 (Volume of a Sphere) Write a program that defines a macro with one argument to compute a sphere's volume. Use the macro to compute the volumes for spheres of radius 1 to 10 and print the results in tabular format. The formula for a sphere's volume is

```
(4.0 / 3) * \pi * r^3
```

where π is 3.14159.

14.5 (Adding Two Numbers) Write a program that defines macro SUM with two arguments, x and y, and use SUM to produce the following output:

```
The sum of x and y is 13
```

14.6 (*Smaller of Two Numbers*) Write a program that defines and uses a macro named MINIMUM2 to determine the smaller of two numeric values.

- **14.7** (Smallest of Three Numbers) Write a program that defines and uses a macro named MINIMUM3 to determine the smallest of three numeric values. Macro MINIMUM3 should use macro MINIMUM2 from Exercise 14.6 to determine the smallest number.
- **14.8** *(Printing a String)* Write a program that defines and uses macro PRINT to print a string value.
- **14.9** (*Printing an Array*) Write a program that defines and uses macro PRINTARRAY to print an array of integers. The macro should receive the array and its number of elements as arguments.
- **14.10** (*Totaling an Array's Contents*) Write a program that defines and uses macro SUMARRAY to sum the values in a numeric array. The macro should receive the array and its number of elements as arguments.

15

Other Topics



Objectives

In this chapter, you'll:

- Write functions that use variable-length argument lists.
- Process command-line arguments.
- Compile multiple-source-file programs.
- Assign specific types to numeric constants.
- Terminate programs with exit and atexit.
- Process external asynchronous events in a program.
- Dynamically allocate arrays and resize memory that was dynamically allocated previously.

15.1	Introduction	15.5	Program Termination with exit
15.2	Variable-Length Argument Lists		and atexit
15.3	Using Command-Line Arguments	15.6	Suffixes for Integer and Floating-
15.4	Compiling Multiple-Source-File		Point Literals
	Programs	15.7	Signal Handling
15.	4.1 extern Declarations for Global Variables in Other Files	15.8	Dynamic Memory Allocation Functions calloc and realloc
	4.2 Function Prototypes		
15.	4.3 Restricting Scope with static	15.9	goto: Unconditional Branching

Summary | Self-Review Exercise | Answers to Self-Review Exercise | Exercises

15.1 Introduction

This chapter presents additional topics not ordinarily covered in introductory courses. Some capabilities discussed here are specific to particular operating systems, especially macOS/Linux and Windows.

15.2 Variable-Length Argument Lists

Most programs in the text have used the standard-library function printf. At a minimum, printf must receive a string as its first argument, but printf can receive any number of additional arguments. The function prototype for printf is

```
int printf(const char *format, ...);
```

The ellipsis (...) in the function prototype indicates that the function receives a *variable number of arguments of any type*. You can use this syntax to define your own functions with variable-length argument lists. The ellipsis must be the last parameter. ERR 🚫 Placing the ellipsis in the middle of the parameter list is a syntax error.

The following table contains the variable arguments (<stdarg.h>) header's macros and definitions for building functions with variable-length argument lists:

Identifier	Explanation
va_list	A type for holding information needed by macros va_start, va_arg and va_end. To access the arguments in a variable-length argument list, an object of type va_list must be defined.
va_start	A macro that you must invoke before accessing a variable-length argument list's arguments. This macro initializes the object declared with va_list for use by the va_arg and va_end macros.
va_arg	A macro that expands to the variable-length argument list's next argument value. The value has the type you specify as the macro's second argument. Each use of va_arg modifies the object declared with va_list to point to the next argument.
va_end	A macro that facilitates a normal return from a function whose variable-length argument list was referred to by the va_start macro.

Figure 15.1 demonstrates a function average (lines 23–36) with a variable-length argument list. The function's first argument is the number of values to average.

```
// fig15_01.c
 2
   // Using variable-length argument lists
 3 #include <stdarq.h>
 4 #include <stdio.h>
 5
 6
    double average(int i, ...); // ... represents variable arguments
 7
 8
    int main(void) {
 9
       double w = 37.5;
       double x = 22.5;
10
11
       double y = 1.7;
12
       double z = 10.2;
13
       printf("%s%.1f; %s%.1f; %s%.1f\n\n",
14
          "W = ", W, "X = ", X, "y = ", y, "Z = ", Z);
15
       printf("%s%.3f\n%s%.3f\n%s%.3f\n",
16
17
          "The average of w and x is ", average(2, w, x),
18
          "The average of w, x, and y is ", average(3, w, x, y),
          "The average of w, x, y, and z is ", average(4, w, x, y, z));
19
20 }
21
    // calculate average
22
23
    double average(int i, ...) {
       double total = 0; // initialize total
24
25
       va_list ap; // stores information needed by va_start and va_end
26
27
       va_start(ap, i); // initializes the va_list object
28
29
       // process variable-length argument list
30
       for (int j = 1; j <= i; ++j) {
31
          total += va_arg(ap, double);
32
33
34
       va_end(ap); // clean up variable-length argument list
35
       return total / i; // calculate average
36 }
w = 37.5; x = 22.5; y = 1.7; z = 10.2
The average of w and x is 30.000
The average of w, x, and y is 20.567
The average of w, x, y, and z is 17.975
```

Fig. 15.1 Using variable-length argument lists.

The average function (lines 23–36) uses all the definitions and macros of header <stdarg.h>, except va_copy (Section C.7.8), which was added in C11. The va_list variable ap (short for "argument pointer"; line 25) is used by macros va_start, va_arg and va_end to process function average's variable-length argument list. First,

the function invokes macro va_start (line 27) to initialize object ap for use by va_arg and va_end. The va_start macro receives:

- the object ap, and
- the identifier of the rightmost argument in the parameter list *before* the ellipsis (i in this example)—va_start uses this argument to determine where the variable-length argument list begins.

Next, the average function repeatedly adds the variable-length argument list's arguments to the variable total (lines 30–32). The macro va_arg retrieves the next value to add to total. The macro receives two arguments:

- the object ap, and
- the value *type* expected in the argument list—double in this case.

The macro returns the argument's value. Line 34 invokes the macro va_end with the object ap as an argument to facilitate a normal return to the caller from average. Finally, line 35 calculates the average and returns it to main.

You might wonder how functions with variable-length argument lists like printf and scanf know what type to use in each va_arg macro call. The answer is that, as the program executes, they scan the format conversion specifiers in the format control string to determine the type of the next argument to process.

\checkmark

Self Check

I (Fill-In) The function prototype for printf is

```
int printf(const char *format, ...);
```

The ellipsis (...) in the prototype indicates that the function receives _____. **Answer:** a variable number of arguments of any type.

- **2** (*Multiple Choice*) Which macro corresponds to the description: "To access the arguments in a variable-length argument list, an object of this type must be defined."
 - a) va_start
 - b) va_end
 - c) va_list
 - d) va_arg

Answer: c.

15.3 Using Command-Line Arguments

Command-line arguments are commonly used to pass options and filenames to a program. The main function may receive arguments from a command line if the function's parameter list contains the parameters int argc and char *argv[]:

- The argc parameter receives the number of command-line arguments that the user has entered.
- The argv parameter is an array of strings containing the command-line arguments.

Figure 15.2 copies a file one character at a time into another file. Assume that the executable file for this program is called mycopy. A typical command line for executing this program is

```
mycopy input output
```

This command line indicates that the file input should be copied to the file output. When the program executes, if argc is not 3 (mycopy counts as one of the arguments), the program prints an error message (line 8) and terminates. Otherwise, the array argv contains the strings "mycopy", "input" and "output". This program uses its second and third command-line arguments as filenames.

```
I // fig15_02.c
 2 // Using command-line arguments
 3 #include <stdio.h>
 4
   int main(int argc, char *argv[]) {
 5
       // check number of command-line arguments
 6
 7
       if (argc != 3) {
 8
          puts("Usage: mycopy infile outfile");
 9
10
       else {
11
          FILE *inFilePtr = NULL; // input file pointer
12
13
          // try to open the input file
          if ((inFilePtr = fopen(argv[1], "r")) != NULL) {
14
             FILE *outFilePtr = NULL; // output file pointer
15
16
17
             // try to open the output file
             if ((outFilePtr = fopen(argv[2], "w")) != NULL) {
18
                int c = 0; // holds characters read from source file
19
20
21
                // read and output characters
22
                while ((c = fgetc(inFilePtr)) != EOF) {
23
                   fputc(c, outFilePtr);
24
25
                fclose(outFilePtr); // close the output file
26
27
28
             else { // output file could not be opened
                printf("File \"%s\" could not be opened\n", argv[2]);
29
             }
30
31
32
             fclose(inFilePtr); // close the input file
33
          }
34
          else { // input file could not be opened
             printf("File \"%s\" could not be opened\n", argv[1]);
35
36
          }
37
       }
   }
38
```

Fig. 15.2 | Using command-line arguments.

We use the function fopen to open these files for reading (line 14) and writing (line 18), respectively. If the program opens both files successfully, lines 22–24 read characters from the file input and write them to the file output. This process continues until the end of input is reached. Then the program terminates. The result is an exact copy of the file input—if no errors occur during processing.

[*Note:* In Visual C++, you specify command-line arguments by right-clicking the project name in the Solution Explorer and selecting Properties, then expanding Configuration Properties, selecting Debugging and entering the arguments in the textbox to the right of Command Arguments.]

Self Check

- I (Fill-In) You can pass command-line arguments to main by including parameters int argc and _____ in main's parameter list.

 Answer: char *argv[].
- **2** (*Discussion*) Assuming inFilePtr represents a successfully opened input file and outFilePtr represents a successfully opened output file, what does the following code segment do?

```
while ((c = fgetc(inFilePtr)) != EOF) {
   fputc(c, outFilePtr);
}
```

Answer: This loop reads one character at a time from the input file and writes it to the output file until the end-of-file indicator for the input file is set.

15.4 Compiling Multiple-Source-File Programs

It's possible to build programs that consist of multiple source files. There are several considerations when creating programs in multiple files. For example, the definition of a function must be entirely contained in one file—it cannot span two or more files.

15.4.1 extern Declarations for Global Variables in Other Files

Chapter 5 introduced storage class and scope concepts. We learned that variables declared *outside* any function definition are *global variables*. Global variables are accessible to any function defined in the same file after the variable is declared. Global variables also can be accessible to functions in other files if they're declared in each file that uses them. For example, to refer to the global integer variable flag in another file, you can use the declaration

```
extern int flag;
```

The storage-class specifier **extern** indicates that flag is defined either *later in the same* file or in a different file. The compiler informs the linker that unresolved references to the variable flag appear in the file. If the linker finds a proper global definition, the linker resolves the references to flag. If the linker cannot locate a definition of flag, it issues an error message and does not produce an executable file. Any identifier that's

declared at file scope is extern by default. You should avoid global variables unless ASE application performance is critical because they violate the principle of least privilege.



15.4.2 Function Prototypes

Just as extern declarations can be used to declare that global variables are defined in other program files, function prototypes can extend the scope of a function beyond the file in which it's defined. The extern specifier is not required in a function prototype. Simply include the function prototype in each file that calls the function, and compile the files together (see Section 14.2). Function prototypes indicate that the specified function is defined either later in the same file or in a different file. Again, the compiler does not resolve references to such a function—the linker performs that task. If the linker cannot locate a proper function definition, the linker issues an error message.

As an example of using function prototypes to extend the scope of a function, consider any program containing the preprocessor directive #include <stdio.h>, which includes a file containing the function prototypes for printf, scanf and many other functions. A file that #includes <stdio.h> can use printf and scanf, even though they're defined in other files. We do not need to know where they're defined. The linker resolves our references to these functions automatically.

Software Reusability

Creating programs in multiple source files facilitates software reusability and good ASE software engineering. Functions that are common to many applications should be stored in their own source files. Each source file should have a corresponding header containing the function prototypes. This enables programmers of different applications to reuse the same code by including the proper header file and compiling their applications with the corresponding source file.



15.4.3 Restricting Scope with static

It's possible to restrict a global variable's or function's scope to the file in which it's defined. Applying the storage-class specifier static to a global variable or function prevents it from being used outside the file that defines it. This is known as internal linkage. Global variables and functions *not* preceded by static in their definitions have external linkage. They can be accessed in other files containing proper declarations.

The global variable definition

```
static const double PI = 3.14159;
```

creates constant variable PI of type double, initializes it to 3.14159 and indicates that PI is known *only* to functions in the file in which it's defined.

The static specifier is commonly used with utility functions called only within a particular file. If a function is not required outside a file, enforce the principle of least privilege by applying static to both the function's definition and prototype.

Self Check

- *(Multiple Choice)* Which of the following statements is *false?*
 - a) Variables declared outside any function definition are global variables.
 - b) Global variables are accessible to any function defined in the same file after the variable is declared.
 - c) Global variables also are accessible to functions in other files.
 - d) Once a global variable is defined, it's known to all the application's files.

Answer: d) is *false*. Actually, global variables must be declared in *each* file in which they're used.

- **2** (*Fill-In*) Any identifier that's declared at file scope is _____ by default. Answer: extern.
- **3** (*True/False*) You should prefer global variables to local variables because global variables enforce the principle of least privilege.

Answer: False. Actually, you should avoid global variables unless application performance is critical because they *violate* the principle of least privilege.

- **4** (*Fill-In*) Applying static to a global variable or a function prevents it from being used by any function that's not defined in the same file—this is called _____ linkage. **Answer:** internal.
- 5 (*True/False*) The following global variable definition indicates that PI is known only to functions in the file in which it's defined:

```
static const double PI = 3.14159;
```

Answer: True.

15.5 Program Termination with exit and atexit

The general utilities library (<stdlib.h>) provides methods of terminating program execution by means other than a conventional return from function main.

exit Function

The exit function terminates a program immediately. This function often is used to terminate a program when an error is detected. The function takes one argument—normally, EXIT_SUCCESS or EXIT_FAILURE: These contain implementation-defined values for successful and unsuccessful termination.

atexit Function

The atexit function registers a function to call when the program terminates by reaching the end of main or when exit is invoked. This function takes as an argument another function's name. Recall that a function name is a pointer to that function. Functions called at program termination cannot have arguments and cannot return a value. When a program terminates, any functions previously registered with atexit are invoked in the reverse order of their registration.

Using Functions exit and atexit

Figure 15.3 tests functions exit and atexit. The program prompts the user to determine whether the program should be terminated with exit or by reaching the end of main. Function print is executed at program termination in each case.

```
// fig15_03.c
 2
    // Using the exit and atexit functions
 3 #include <stdio.h>
  #include <stdlib.h>
 4
 5
 6
   void print(void); // prototype
 7
 8
    int main(void) {
 9
       atexit(print); // register function print
10
       puts("Enter 1 to terminate program with function exit\n"
П
            "Enter 2 to terminate program normally");
       int answer = 0; // user
12
       scanf("%d", &answer);
13
14
15
       // call exit if answer is 1
16
       if (answer == 1) {
          puts("\nTerminating program with function exit");
17
          exit(EXIT_SUCCESS);
18
19
20
       puts("\nTerminating program by reaching the end of main");
21
22
   }
23
24
   // display message before termination
    void print(void) {
25
26
       puts("Executing function print at program termination\n"
27
            "Program terminated");
28
   }
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
Terminating program with function exit
Executing function print at program termination
Program terminated
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally

2

Terminating program by reaching the end of main
Executing function print at program termination
Program terminated
```

Fig. 15.3 Using the exit and atexit functions.

Self Check

I (*True/False*) Function atexit terminates a program immediately.

Answer: *False.* Actually, function exit causes a program to terminate immediately. Function atexit registers a function to call when a program terminates by reaching the end of main or when exit is invoked.

2 (*TruelFalse*) Calling exit with EXIT_SUCCESS returns 1 to the calling environment, and calling exit with EXIT_FAILURE returns 0.

Answer: False. Actually, calling exit with EXIT_SUCCESS or EXIT_FAILURE returns implementation-defined values for successful or unsuccessful termination.

15.6 Suffixes for Integer and Floating-Point Literals

Integer and floating-point *suffixes* enable you to specify explicitly the data types of literal values. By default, an integer literal's type is determined by the first type capable of storing the value—int, then long int, then unsigned long int, etc. A floating-point literal with no suffix has type double.

The integer suffixes are u or U for unsigned ints, 1 or L for long ints, and 11 or LL for long long ints. L and LL are preferred for readability, as a lowercase 1 can be mistaken as a 1 (one). You can combine u or U with those for long int and long long int to create unsigned literals for the larger integer types. The following literals have types unsigned int, long int, unsigned long int and unsigned long long int:

```
174u
8358L
28373ul
9876543210]]u
```

The floating-point suffixes are f or F for floats, and l or L for long doubles. Again, L is preferred for readability. The following are float and long double literals:

```
1.28f
3.14159L
```

Self Check

I (Fill-In) C provides integer and floating-point _____ for explicitly specifying the types of integer and floating-point literal values.

Answer: suffixes.

2 (*Fill-In*) The following constants have types _____ and ____.

```
1.28f
3.14159L
```

Answer: float, long double.

15.7 Signal Handling

An external asynchronous event, or signal, can cause a program to terminate prematurely. Some events include:

- interrupts, such as typing *<Ctrl> c* (Linux or Windows) or *<command> c* (macOS), and
- termination orders from the operating system.

The signal-handling library (<signal.h>) enables programs to trap unexpected events with function signal, which receives two arguments:

- an integer signal number, and
- a pointer to a signal-handling function.

A program can generate signals by calling function **raise**, which takes an integer signal number as an argument. The following table summarizes the *standard signals* from the <signal.h> header:

Signal	Explanation
SIGABRT	Abnormal termination of the program (such as a call to function abort).
SIGFPE	An erroneous arithmetic operation, such as a divide-by-zero or an operation resulting in overflow.
SIGILL	Detection of an illegal instruction.
SIGINT	Receipt of an interactive attention signal (<i>Ctrl> c</i> or <i><command/> c</i>).
SIGSEGV	An attempt to access memory that is not allocated to a program.
SIGTERM	A termination request sent to the program.

Demonstrating Signal Handling

Figure 15.4 uses function signal to *trap* a SIGINT. Line 11 calls signal with SIGINT and a pointer to the function signalHandler. When a SIGINT signal occurs, control passes to function signalHandler, which prints a message and gives the user the option to continue normal program execution. If the user wishes to continue execution, line 49 reinitializes the signal handler by calling signal again, and control returns to the point in the program at which the signal was detected.

```
// fig15_04.c
2 // Using signal handling
3 #include <signal.h>
4 #include <stdio.h>
5 #include <stdlib.h>
   #include <time.h>
7
8
   void signalHandler(int signalValue); // prototype
9
10
   int main(void) {
       signal(SIGINT, signalHandler); // register signal handler
11
12
       srand(time(NULL));
13
```

Fig. 15.4 Using signal handling. (Part 1 of 3.)

```
// output numbers 1 to 100
14
15
       for (int i = 1; i \le 100; ++i) {
16
          int x = 1 + rand() \% 50; // generate random number to raise SIGINT
17
          // raise SIGINT when x is 25
18
19
          if (x == 25) {
20
             raise(SIGINT);
21
22
23
          printf("%4d", i);
24
25
          // output \n when i is a multiple of 10
26
          if (i % 10 == 0) {
             printf("%s", "\n");
27
          }
28
29
       }
30
    }
31
32
    // handles signal
33
    void signalHandler(int signalValue) {
34
       printf("\n%s%d%s\n%s",
          "Interrupt signal (", signalValue, ") received.",
35
          "Do you wish to continue (1 = yes or 2 = no)?");
36
37
       int response = 0; // user
       scanf("%d", &response);
38
39
40
       // check for invalid responses
41
       while (response != 1 && response != 2) {
          printf("%s", "(1 = yes or 2 = no)? ");
42
43
          scanf("%d", &response);
       }
44
45
46
       // determine whether to continue
47
       if (response == 1) {
          // reregister signal handler for next SIGINT
48
          signal(SIGINT, signalHandler);
49
50
       }
51
       else {
52
          exit(EXIT_SUCCESS);
53
       }
54
   }
                  5
           3
               4
                       6
                          7
                                8
                                    9
                                      10
   1
       2
     12
          13 14
                  15 16 17 18 19
                                       20
  11
  21
      22
          23
              24
                  25
                      26
                         27 28
                                   29
                                       30
  31 32
         33 34 35
                      36
Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 1
  37
     38
         39
              40
  41 42
         43 44
                  45
                      46 47
                              48
                                  49
                                       50
                          57
  51 52
          53
              54
                  55
                      56
                              58
                                   59
                                       60
  61 62
          63
              64
                  65
                      66
                          67
                              68
                                   69
                                      70
```

Fig. 15.4 Using signal handling. (Part 2 of 3.)

```
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92
Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 2
```

Fig. 15.4 Using signal handling. (Part 3 of 3.)

In this program, function raise simulates a SIGINT. We choose a random number between 1 and 50. If the number is 25, line 20 calls raise to generate the signal. Normally, SIGINTs are initiated outside the program when someone types <Ctrl> c (Linux or Windows) or <command> c (macOS) to terminate program execution. Signal handling can be used to trap the SIGINT and prevent the program from terminating.

Self Check

I (*Fill-In*) An external asynchronous event, or ______, can cause a program to terminate prematurely.

Answer: signal.

- **2** (*Multiple Choice*) Which standard signal is described by: "An erroneous arithmetic operation, such as a divide-by-zero or an operation resulting in overflow."
 - a) SIGILL
 - b) SIGABRT
 - c) SIGINT
 - d) SIGFPE

Answer: d.

15.8 Dynamic Memory Allocation Functions calloc and realloc

Chapter 12 introduced the notion of dynamically allocating memory using function malloc. As we stated in Chapter 12, arrays are better than linked lists for rapid sorting, searching and data access. Arrays are normally static data structures that cannot be resized. The general utilities library (<stdlib.h>) provides dynamic memory allocation functions calloc and realloc to create dynamic arrays and modify their sizes.

calloc Function

The calloc ("contiguous allocation") function

void *calloc(size_t nmemb, size_t size);

dynamically allocates an array. Its two arguments are

- the array's number of elements (nmemb), and
- each element's size (size).

Function calloc also initializes the array's elements to zero. The function returns a pointer to the allocated memory or a NULL pointer if it could not allocate the memory. The primary difference between malloc and calloc is that calloc clears the memory it allocates, whereas malloc does not.

realloc Function

The realloc function

```
void *realloc(void *ptr, size_t size);
```

changes the size of an object allocated by a previous malloc, calloc or realloc call. The original object's contents are not modified as long as the amount of memory allocated is larger than the amount allocated previously. Otherwise, the contents are unchanged up to the new object's size. The functions two arguments are

- a pointer to the original object (ptr), and
- the object's new size (size).

If ptr is NULL, realloc works identically to malloc. If ptr is not NULL and size is greater than zero, realloc tries to allocate a new block of memory for the object. If the new space cannot be allocated, the object pointed to by ptr is unchanged. Function realloc returns either a pointer to the reallocated memory or a NULL pointer to indicate that the memory was not reallocated.

Self Check

- I (Fill-In) The general utilities library (<std1ib.h>) dynamic memory allocation functions _____ and ____ create and modify dynamic arrays.

 Answer: calloc and realloc.
- 2 (Multiple Choice) Which of the following statements is false?
 - a) Function calloc dynamically allocates memory for an array.
 - b) Function calloc's parameters size_t nmemb and size_t size represent the new array's number of elements and each element's size.
 - c) Function calloc initializes a dynamically allocated array's elements to zero. The function returns a pointer to the allocated memory, or NULL if the memory could not be allocated.
 - d) Functions malloc and calloc clear the memory they allocate.

Answer: d) is *false*. Actually, calloc clears the memory it allocates, but malloc does not.

3 (*TruelFalse*) If realloc's first argument is NULL, it works identically to malloc. Otherwise, if realloc's size argument is greater than zero, it tries to allocate a new block of memory. If it cannot be allocated, the object pointed to by the function's first argument is unchanged. The function returns either a pointer to the reallocated memory or a NULL pointer to indicate that the memory was not reallocated.

Answer: True.

15.9 goto: Unconditional Branching

We've stressed the importance of using structured-programming techniques to build reliable software that's easy to debug, maintain and modify. In some cases, performance is more important than strict adherence to structured-programming techniques. In these cases, some unstructured-programming techniques may be used. For example, we can use break to terminate an iteration statement's execution before the loop-continuation condition becomes false. This saves unnecessary iterations of the loop if the task is completed *before* loop termination.

Another instance of unstructured programming is the **goto** statement—an unconditional branch. The goto statement alters the flow of control, continuing execution with the first statement after the label specified in the statement. A label is an identifier followed by a colon (:). A label must appear in the *same* function as the goto statement that refers to it. Labels need not be unique among functions.

Demonstrating goto

Figure 15.5 uses goto statements to loop ten times and print a counter value each time. Line 6 initializes count to 1. The label start: is skipped, because labels do not perform any action. Line 9 tests whether count is greater than 10. If so, line 10 transfers control from the goto to the first statement after the label end: (line 19). Otherwise, lines 13–14 print and increment count, and control transfers from the goto (line 16) to the first statement after the label start: (line 9).

```
// fig15_05.c
 2
   // Using the goto statement
 3
   #include <stdio.h>
 4
 5
    int main(void) {
 6
       int count = 1; // initialize count
 7
 8
       start: // label
 9
          if (count > 10) {
10
             goto end;
П
12
13
          printf("%d ", count);
14
          ++count;
15
          goto start; // goto start on line 9
16
17
18
       end: // label
          putchar('\n');
19
20
    }
  2 3 4 5 6 7 8 9 10
1
```

Fig. 15.5 Using the **goto** statement.

714 Chapter 15 Other Topics

Chapter 3 stated you can write any program in terms of sequence, selection and iteration statements. When following the structured-programming rules, you can create deeply nested control structures within a function from which it's difficult to PERF : escape efficiently. Some programmers use goto statements in such situations as a quick exit from a deeply nested structure. This eliminates the need to test multiple conditions to escape from a control structure. There are other situations where goto is actually recommended—see, for example, CERT recommendation MEM12-C, "Consider using a Goto-Chain when leaving a function on error when using and releasing resources." Note that the goto statement is unstructured and can lead to SE A programs that are more difficult to debug, maintain and modify.

Self Check

- [Multiple Choice] Which of the following statements a), b) or c) is false?
 - a) Structured-programming techniques help you to build reliable software that's easy to debug, maintain and modify.
 - b) In some cases, performance is more important than strict adherence to structured-programming techniques. In these cases, you might choose to use some unstructured-programming techniques.
 - c) We can use break to terminate an iteration statement early. This saves unnecessary iterations of the loop if the task is completed before loop termination.
 - d) All of the above statements are *true*.

Answer: d.

- **2** (*Multiple Choice*) Which of the following statements a), b) or c) is *false?*
 - a) An instance of unstructured programming is the goto statement—an unconditional branch.
 - b) The goto statement alters the flow of control by continuing execution with the first statement after the label specified in the statement. A label is an identifier followed by a colon.
 - c) Labels must be unique among all the functions in an application. A label that's the target of a particular goto statement may appear in any function in an application.
 - d) All of the above statements are true.

Answer: c) is *false*. Actually, labels need not be unique among functions. Also, a label that's the target of a goto statement in a function must appear in that function.

(True/False) When you follow the rules of structured programming, it's possible to create deeply nested control structures within a function from which it's difficult to escape efficiently. Some programmers use goto statements in such situations as a quick exit from a deeply nested structure. This eliminates the need to test multiple conditions to escape from a control structure.

Answer: True.

Summary

Section 15.2 Variable-Length Argument Lists

- The header <stdarg.h> (p. 700) provides capabilities for building functions with variable-length argument lists.
- An ellipsis (...; p. 700) in a function prototype indicates a variable number of arguments.
- A va_list (p. 701) holds information needed by macros va_start, va_arg and va_end.
- Macro va_start (p. 701) initializes a va_list object for use by the va_arg and va_end.
- Macro va_arg (p. 701) expands to the value and type of the variable-length argument list's next argument. Each invocation of va_arg modifies the object declared with va_list to point to the next argument.
- Macro va_end (p. 701) facilitates a normal return from a function whose variable-length argument list was referred to by the va_start macro.

Section 15.3 Using Command-Line Arguments

• To pass arguments to main from the command line, include the parameters int argc and char *argv[] (p. 702) in main's parameter list. Parameter argc receives the number of command-line arguments. Parameter argv is an array of strings in which the command-line arguments are stored.

Section 15.4 Compiling Multiple-Source-File Programs

- A function definition must be entirely contained in one file.
- The storage-class specifier extern (p. 704) indicates that a variable is defined either later in the same file or in a different file of the program.
- Global variables must be declared in each file in which they're used.
- A function prototype can extend a function's scope beyond the file in which it's defined.
- Applying storage-class specifier static to a global variable or function prevents it from being used outside the current file. This is called internal linkage (p. 705). Global variables and functions not preceded by static have external linkage (p. 705) and can be accessed in other files if those files contain proper declarations or function prototypes.
- The static specifier is commonly used with utility functions that are called only by functions in a particular file.
- If a function is not required outside a particular file, apply static to it to enforce the principle of least privilege.

Section 15.5 Program Termination with exit and atexit

- Function **exit** (p. 706) forces a program to terminate.
- Function atexit (p. 706) registers a function to call when the program terminates by reaching the end of main or when exit is invoked.
- Function atexit takes a pointer to a function as an argument. Functions called at program termination cannot have arguments and cannot return a value.
- Function exit takes one argument, normally the symbolic constant **EXIT_SUCCESS** (p. 706) or the symbolic constant **EXIT_FAILURE** (p. 706).
- When function exit is invoked, any functions registered with atexit are invoked in the reverse order of their registration.

Section 15.6 Suffixes for Integer and Floating-Point Literals

• Integer and floating-point suffixes can be used to specify the types of integer and floating-point constants. The integer suffixes are u or U for an unsigned integer, 1 or L for a long integer, and ull or UL for an unsigned long integer. The type of an integer constant with no suffix is determined by the first type capable of storing a value of that size (int, then long int, then unsigned long int, etc.). The floating-point suffixes are f or F for a float, and l or L for a long double. A floating-point constant with no suffix has type double.

Section 15.7 Signal Handling

- The **signal-handling library** (p. 709) enables trapping of unexpected events with function **signal** (p. 709).
- Function signal receives two arguments—an integer signal number and a pointer to the signal-handling function.
- Signals can also be generated with function raise (p. 709) and an integer argument.

Section 15.8 Dynamic Memory Allocation: Functions calloc and realloc

- The **general utilities library** (**stdlib.h**>) provides dynamic memory allocation functions calloc and realloc for creating dynamic arrays and resizing them.
- Function **calloc** (p. 711) allocates memory for an array. It receives the array's number of elements and each element's size and initializes the array's elements to zero. It returns either a pointer to the allocated memory or a NULL pointer if the memory is not allocated.
- Function **realloc** changes the size of an object allocated by a previous malloc, calloc or realloc call. The original object's contents are not modified as long as the amount of memory allocated is larger than the amount allocated previously.
- Function realloc receives a pointer to the original object and the new size of the object. If ptr is NULL, realloc works identically to malloc. Otherwise, if ptr is not NULL and size is greater than zero, realloc tries to allocate a new block of memory for the object. If the new space cannot be allocated, the object pointed to by ptr is unchanged. Function realloc returns either a pointer to the reallocated memory or a NULL pointer to indicate that memory was not reallocated.

Section 15.9 Unconditional Branching with goto

- The **goto** statement (p. 713) alters a program's flow of control. Program execution continues at the first statement after the label (p. 713) specified in the goto statement.
- A label is an **identifier followed by a colon**. A label must appear in the same function as the goto statement that refers to it.

Self-Review Exercise

Jell.	-Keview Laeicise	
15.1	Fill-In the blanks in each of the following:	
	a) A(n) in the parameter list of a function indicates that the	function
	can receive a variable number of arguments.	
	b) Macro must be invoked before the arguments in a variable	le-length
	argument list can be accessed.	
	c) Macro accesses the individual arguments of a variable-leng	gth argu-
	ment list.	

d)	Macro facilitates a normal return from a function whose va	ıriable-
	length argument list was referred to by macro va_start.	
e)	Argument of main receives the number of arguments in a	a com-
	mand line.	
f)	Argument of main stores command-line arguments as ch	aracter
	strings.	
g)	Function forces a program to terminate execution.	
h)	Function registers a function to be called upon normal pr	rogram
	termination.	
i)	An integer or floating-point can be appended to an integer o	r float-
	ing-point constant to specify the exact type of the constant.	
j)	Function can be used to trap unexpected events.	
k)	Function generates a signal from within a program.	
1)	Function dynamically allocates memory for an array and ini	tializes
	the elements to zero.	
m	Function changes the size of a block of previously allocate	ted dy-
	namic memory.	

Answers to Self-Review Exercise

15.1 a) ellipsis (...). b) va_start. c) va_arg. d) va_end. e) argc. f) argv. g) exit. h) atexit. i) suffix. j) signal. k) raise. l) calloc. m) realloc.

Exercises

- **15.2** (Variable-Length Argument List: Calculating Products) Write a program that calculates the product of a series of integers that are passed to function product using a variable-length argument list. Test your function with several calls, each with a different number of arguments.
- **15.3** (*Printing Command-Line Arguments*) Write a program that prints the command-line arguments of the program.
- **15.4** (Sorting Integers) Write a program that sorts an array of integers into ascending or descending order. Use command-line arguments to pass either argument -a for ascending order or -d for descending order. [Note: This is the standard format for passing options to a program in UNIX.]
- **15.5** (Signal Handling) Read the documentation for your compiler to determine which signals are supported by the signal-handling library (<signal.h>). Write a program that contains signal handlers for the standard signals SIGABRT and SIGINT. The program should trap these signals by calling function abort to generate a signal of type SIGABRT and by having the user type <*Ctrl> c* or <*command> C* to generate a signal of type SIGINT.
- **15.6** (*Dynamic Array Allocation*) Write a program that dynamically allocates an array of integers. The size of the array should be input from the keyboard. The elements

718 Chapter 15 Other Topics

of the array should be assigned values input from the keyboard. Print the array's values. Next, reallocate the memory for the array to half of the current number of elements. Print the array's remaining values to confirm they match the first half of the original array's values.

15.7 (Command-Line Arguments) Write a program that takes two filenames as command-line arguments, reads the first file's characters one at a time and writes them to the second file in reverse order.



A

Operator Precedence Chart

Operators are shown in decreasing order of precedence from top to bottom.

		*
Operator	Туре	Associativity
0	parentheses (function-call operator)	left to right
[]	array subscript	_
	member selection via object	
->	member selection via pointer	
++	unary postincrement	
	unary postdecrement	
++	unary preincrement	right to left
	unary predecrement	
+	unary plus	
-	unary minus	
!	unary logical negation	
~	unary bitwise complement	
(type)	C-style unary cast	
*	dereference	
&	address	
sizeof	determine size in bytes	1.6 . 1
*	multiplication	left to right
/	division	
%	modulus	
+	addition	left to right
-	subtraction	
<<	bitwise left shift	left to right
>>	bitwise right shift	
<	relational less than	left to right
<=	relational less than or equal to	
>	relational greater than	
>=	relational greater than or equal to	
==	relational is equal to	left to right
!=	relational is not equal to	
&	bitwise AND	left to right

Operator	Туре	Associativity
٨	bitwise exclusive OR	left to right
1	bitwise inclusive OR	left to right
&&	logical AND	left to right
П	logical OR	left to right
?:	ternary conditional	right to left
=	assignment	right to left
+=	addition assignment	
-=	subtraction assignment	
*=	multiplication assignment	
/=	division assignment	
%=	modulus assignment	
&=	bitwise AND assignment	
Λ=	bitwise exclusive-OR assignment	
=	bitwise inclusive-OR assignment	
<<=	bitwise left-shift assignment	
>>=	bitwise right-shift assignment	
,	comma	left to right



B

ASCII Character Set

In the following table, the digits in the left column are the left digits of the character code's decimal equivalent (0–127), and the digits in the top row are the right digits of the character code's decimal equivalent. For example, the character code for "A" in row number 6 and column number 5 is 65, and the character code for "&" in row number 3 and column number 8 is 38.

ASCI	II Charac	cter Set								
	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	1f	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	6
4	()	*	+	,	-		/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	Α	В	C	D	Е
7	F	G	Н	I	J	K	L	M	N	0
8	Р	Q	R	S	Т	U	V	W	Χ	Υ
9	Z	[\]	٨	_	,	a	b	С
10	d	е	f	g	h	i	j	k	1	m
11	n	0	р	q	r	S	t	u	V	W
12	X	У	Z	{	1	}	~	del		

Multithreading/Multicore and Other C18/C11/C99 Topics





Objectives

In this appendix, you'll:

- Understand the purpose of C18.
- Learn the headers added in C99 and C11/C18.
- Initialize arrays and structs with designated initializers.
- Use data type bool to create boolean variables whose data values can be true or false.
- Perform arithmetic operations on complex variables.
- Learn about preprocessor enhancements.
- Use multithreading to improve performance on today's multicore systems.

C.1 Introduction	C.8 CII/CI8 Features
C.2 Headers Added in C99	C.8.1 CII/CI8 Headers
C.3 Designated Initializers and Compound Literals	C.8.2 quick_exit Function C.8.3 Unicode® Support C.8.4 _Noreturn Function Specifier
C.4 Type boolC.5 Complex Numbers	C.8.5 Type-Generic Expressions C.8.6 Annex L: Analyzability and Undefined Behavior
C.6 Macros with Variable-Length Argument Lists	C.8.7 Memory Alignment Control C.8.8 Static Assertions C.8.9 Floating-Point Types
C.7 Other C99 Features C.7.1 Compiler Minimum Resource Limits C.7.2 The restrict Keyword C.7.3 Reliable Integer Division	C.9 Case Study: Performance with Multithreading and Multicore Systems
C.7.4 Flexible Array Members C.7.5 Type-Generic Math C.7.6 Inline Functions C.7.7 Func Predefined Identifier	C.9.1 Example: Sequential Execution of Two Compute-Intensive Tasks C.9.2 Example: Multithreaded Execution of Two Compute-Intensive Tasks

C. I Introduction

C.7.8 va_copy Macro

C.7.7 __func__ Predefined Identifier

The C99 (1999) and C11 (2011) standards refined and expanded C's capabilities. Since C11, there has been only one new version, C18¹ (2018). It "addressed defects in C11 without introducing new language features." Some features added by the C99 and C11/C18 standards are designated as optional. Before using the features shown in this appendix, check that your compiler supports them. Our goal is to introduce these capabilities and provide resources for further reading.

C.9.3 Other Multithreading Features

We explain with complete code examples and code snippets designated initializers, compound literals, type bool and complex numbers. We provide brief explanations of additional features, including restricted pointers, reliable integer division, flexible array members, generic math, inline functions and return without expression.

We discuss C11/C18 capabilities, including improved Unicode[®] support, the function specifier _Noreturn, type-generic expressions, the quick_exit function, memory alignment control, static assertions, analyzability and floating-point types.

C11/C18 Multithreading

A key feature of this appendix is Section C.9's introduction to multithreading. In today's multicore systems, the hardware can put multiple processors (cores) to work on different parts of your task. This enables the tasks (and the program) to complete faster. To take advantage of multicore architecture from C programs, you need to write multithreaded applications. When a program splits tasks into separate threads,

ISO/IEC 9899:2018, Information technology — Programming languages — C, https:// www.iso.org/standard/74528.html.

^{2.} https://en.wikipedia.org/wiki/C18_(C_standard_revision). Also http://www.iso-9899.info/wiki/The_Standard.

a multicore system can run those threads in parallel—that is, simultaneously. Section C.9 first demonstrates two long-running calculations performed in sequence. Then we separate those calculations into two threads to demonstrate the significant performance improvement of running the threads in parallel on multiple cores.

C.2 Headers Added in C99

The following table lists the standard-library headers added in C99—these remain available in C11/C18. We'll discuss the new C11/C18 headers in Section C.8.1.

Header	Explanation
<complex.h></complex.h>	Contains support for complex numbers (see Section C.5).
<fenv.h></fenv.h>	Provides information about the C implementation's floating-point environment and capabilities.
<inttypes.h></inttypes.h>	Defines portable integral types and provides format specifiers for them.
<stdbool.h></stdbool.h>	Contains macros defining bool, true and false, used for boolean variables (see Section C.4).
<stdint.h></stdint.h>	Defines extended integer types and related macros.
<tgmath.h></tgmath.h>	Provides type-generic macros that allow functions from <math.h> to be used with a variety of parameter types (see Section C.7).</math.h>

C.3 Designated Initializers and Compound Literals

[This section can be read after Section 10.3.]

Designated initializers allow you to initialize array elements by subscript and union or struct members by name. Figure C.1 shows that we can use designated initializers to initialize specific array elements.

```
// figC_01.c
2
   // Initializing specific array elements with designated initializers.
3
   #include <stdio.h>
4
5
   int main(void) {
6
       int values[5] = {
7
          [0] = 123, // initialize element 0
          [4] = 456 // initialize element 4
8
9
       }; // semicolon is required
10
       // output array contents
П
       printf("values: ");
12
13
       for (size_t i = 0; i < 5; ++i) {
14
          printf("%d ", values[i]);
15
       }
16
```

Fig. C.1 | Initializing specific array elements with designated initializers. (Part 1 of 2.)

```
17
18 puts("");
19 }
values: 123 0 0 0 456
```

Fig. C.1 Initializing specific array elements with designated initializers. (Part 2 of 2.)

Lines 6–9 define an array and initialize its elements 0 and 4 within the braces. Note the syntax. You separate each initializer from the next by a comma. The initializer list's end brace must be followed by a semicolon. Elements that are not explicitly initialized are implicitly initialized to zero.

Compound Literals

You can use an initializer list to create an unnamed array, struct or union. This is known as a compound literal. For example, you could pass Fig. C.1's array to a function without having to declare it beforehand, as in

```
demoFunction((int [5]) {[0] = 1, [4] = 2});
```

Figure C.2 uses compound literals as designated initializers for specific elements in an array of structs. Lines 12 and 13 each use a designated initializer to explicitly initialize a struct element in the array. For example, in line 12, the following expression is a compound literal that creates an anonymous struct object of type struct twoInt:

```
\{.x = 1, .y = 2\}
```

That object's x and y members are initialized to 1 and 2. Designated initializers for struct and union members list each member's name preceded by a dot (.).

```
// figC_02.c
 1
   // Initializing struct members with designated initializers.
 2
   #include <stdio.h>
 3
 5
    struct twoInt { // declare a struct of two integers
 6
       int x;
 7
       int y;
 8
    };
 9
10
    int main(void) {
       struct twoInt a[5] = {
11
          [0] = \{.x = 1, .y = 2\},
12
13
          [4] = \{.x = 10, .y = 20\}
       };
14
15
16
       // output array contents
       printf("%2s%5s\n", "x", "y");
17
18
```

Fig. C.2 | Initializing struct members with designated initializers. (Part 1 of 2.)

```
19
        for (size_t i = 0; i < 5; ++i) {
           printf("\frac{2d}{5d}", a[i].x, a[i].y);
20
        }
21
22
    }
 Х
       У
 1
       2
 0
       0
 0
       0
 0
       0
10
      20
```

Fig. C.2 | Initializing struct members with designated initializers. (Part 2 of 2.)

Lines 11–14 are more straightforward than following executable code, which does not use designated initializers:

```
struct twoInt a[5];
a[0].x = 1;
a[0].y = 2;
a[4].x = 10;
a[4].y = 20;
```

Using initializers rather than runtime assignments improves program startup time.

₹PERF

C.4 Type bool

[This section can be read after Section 3.6.]

The boolean type—_Bool—can hold only the values 0 or 1. Recall that in C conditions, zero represents *false*, and any nonzero value represents *true*. Assigning *any* nonzero value to a _Bool sets it to 1. The <stdbool.h> header defines macros bool, false and true. These macros replace bool with the keyword _Bool, false with 0, and true with 1. Figure C.3 uses a function named isEven (lines 28–35) that returns the bool value true if the function's argument is even and false if it's odd.

```
// figC_03.c
1
   // Using bool, true and false.
2
   #include <stdio.h>
   #include <stdbool.h> // allows the use of bool, true, and false
4
5
    bool isEven(int number); // function prototype
6
7
    int main(void) {
8
       // loop for 2 inputs
9
       for (int i = 0; i < 2; ++i) {
10
          printf("Enter an integer: ");
П
          int input = 0; // value entered by user
12
          scanf("%d", &input);
13
14
```

Fig. C.3 | Using bool, true and false. (Part I of 2.)

```
bool valueIsEven = isEven(input); // determine if input is even
15
16
          // determine whether input is even
17
          if (valueIsEven) {
18
             printf("%d is even\n\n", input);
19
          }
20
21
          else {
             printf("%d is odd\n\n", input);
22
          }
23
       }
24
    }
25
26
    // isEven returns true if number is even
27
28
    bool isEven(int number) {
       if (number \% 2 == 0) \{ // is number divisible by 2?
29
30
          return true;
31
       }
32
       else {
          return false;
33
       }
34
35 }
Enter an integer: 34
34 is even
Enter an integer: 23
23 is odd
```

Fig. C.3 | Using bool, true and false. (Part 2 of 2.)

Line 15 declares the bool variable valueIsEven and passes the user's input to function isEven, which returns a bool value. Line 29 determines whether the argument is divisible by 2. If so, line 30 returns true; otherwise, line 33 returns false. The result is assigned to bool variable valueIsEven in line 15. If valueIsEven is true, line 19 displays a string indicating that the value is even. If valueIsEven is false, line 22 displays a string indicating that the value is odd. Function isEven's body can be written more concisely as

```
return number % 2 == 0;
```

but we wanted to demonstrate the <stdbool.h> header's true and false macros.

C.5 Complex Numbers

[This section can be read after Section 5.3.]

C99 introduced support for complex numbers and complex arithmetic. Figure C.4 shows basic complex-number operations. We compiled and ran this program using Apple's Xcode. Microsoft Visual C++ supports only the complex-number features defined by the C++ standard, not those from C.

```
// figC_04.c
    // Complex number operations.
   #include <complex.h> // for complex type and math functions
 3
 4
   #include <stdio.h>
 5
 6
    int main(void) {
 7
        double complex a = 3.0 + 2.0 * I;
        double complex b = 2.7 + 4.9 * I;
 8
 9
        printf("a is %.1f + %.1fi\n", creal(a), cimag(a));
printf("b is %.1f + %.1fi\n", creal(b), cimag(b));
10
11
12
        double complex sum = a + b; // perform complex addition
13
        printf("a + b is: \%.1f + \%.1fi\n", creal(sum), cimag(sum));
14
15
16
        double complex difference = a - b; // perform complex subtraction
        printf("a - b is: %.1f + %.1fi\n", creal(difference), cimag(difference));
17
18
19
        double complex product = a * b; // perform complex multiplicaton
20
        printf("a * b is: \%.1f + \%.1fi \setminus n", creal(product), cimag(product));
21
22
        double complex quotient = a / b; // perform complex division
23
        printf("a / b is: %.1f + %.1fi\n", creal(quotient), cimag(quotient));
24
25
        double complex power = \frac{\text{cpow}(a, 2.0)}{\text{cpow}(a, 2.0)}; // perform complex exponentiation
26
        printf("a \land b is: %.1f + %.1fi\n", creal(power), cimag(power));
27
    }
a is 3.0 + 2.0i
b is 2.7 + 4.9i
a + b is: 5.7 + 6.9i
a - b is: 0.3 + -2.9i
a * b is: -1.7 + 20.1i
a / b is: 0.6 + -0.3i
a ^ b is: 5.0 + 12.0i
```

Fig. C.4 | Complex number operations.

To use **complex** numbers, include the **complex.h**> header (line 3). This will expand the macro complex to the keyword **_Complex**—a type that reserves an array of exactly two elements, corresponding to the complex number's *real part* and *imaginary part*. You define complex variables as shown in lines 7, 8, 13, 16, 19, 22 and 25. We define each variable as double complex, indicating that the complex number's real and imaginary parts are stored as double values. C also supports float complex or long double complex.

The arithmetic operators work with complex numbers and the <complex.h> header provides additional math functions, such as cpow in line 25. You can also use the operators !, ++, --, &&, ||, ==, != and unary & with complex numbers.

Lines 13–26 output the results of various arithmetic operations. You access a complex number's *real part* and *imaginary part* via functions **creal** and **cimag**, respectively, as shown in line 10. In line 26's output, we use the symbol ^ to indicate exponentiation.

C.6 Macros with Variable-Length Argument Lists

Macros may have variable-length argument lists. This allows for macro wrappers around functions like printf. For example, to automatically add the name of the current file to a debug statement, you can define a macro as follows:

```
#define DEBUG(...) printf(__FILE__ ": " __VA_ARGS__)
```

This DEBUG macro takes a variable number of arguments, as indicated by the ... in the argument list. As with functions, the ... must be the last argument. Unlike functions, the ... can be the macro's *only* argument. The identifier __VA_ARGS__, which begins and ends with two underscores, is a placeholder for the variable-length argument list. Assuming this macro appears in the file file.c, the preprocessor replaces the following macro call:

```
DEBUG("x = %d, y = %d\n", x, y);
with
printf("file.c" ": " "x = %d, y = %d\n", x, y);
```

Recall that strings separated by whitespace are concatenated during preprocessing, so the three string literals will be combined to form printf's first argument.

C.7 Other C99 Features

Here we provide brief overviews of some additional C99 features. These include keywords, language capabilities and standard-library additions.

C.7.1 Compiler Minimum Resource Limits

[This section can be read after Section 15.4.]

Before C99, the standard required C implementations to support identifiers of

- no less than 31 characters for identifiers with internal linkage, and
- no less than six characters for identifiers with external linkage (Section 15.4).

C99 increased these limits to 63 characters for identifiers with internal linkage and 31 characters for identifiers with external linkage. These are just lower limits. Compilers are free to support identifiers with more characters than these limits. For more information, see the C18 standard's Section 5.2.4.1.

C also sets minimum limits on many language features. For example, compilers must support at least 1,023 members in a struct, enum or union, and at least 127 parameters to a function. For more information on other limits, see the C18 Standard Section 5.2.4.1.

C.7.2 The restrict Keyword

[This section can be read after Section 7.5.]

The keyword restrict declares a restricted pointer that should have exclusive access to a region of memory. Objects accessed through a restricted pointer cannot be

accessed by other pointers, except when those pointers' values are derived from the restricted pointer's value, e.g., by assigning a restrict qualified pointer to a nonrestrict qualified pointer.

We can declare a restricted pointer to an int as:

```
int *restrict ptr;
```

Restricted pointers allow the compiler to optimize the way the program accesses memory. For example, the standard-library function memcpy is defined as follows:

```
void *memcpy(void *restrict s1, const void *restrict s2, size_t n);
```

The memcpy function's specification states that it should not be used to copy between overlapping regions of memory. Using restricted pointers allows the compiler to optimize the copy operation by copying multiple bytes simultaneously, which is more 📑 PERF efficient. Incorrectly declaring a pointer as restricted when another pointer points to the same region of memory can result in *undefined behavior*. For more information, see C99 Standard Section 6.7.3.1.

C.7.3 Reliable Integer Division

[This section can be read after Section 2.5.]

In early C compilers, integer division behaviors varied across implementations. Some rounded a negative quotient toward negative infinity, while others rounded toward zero, resulting in different answers. Consider dividing -28 by 5. The exact answer is -5.6. If we round the quotient toward zero, we get -5. If we round -5.6 toward negative infinity, we get -6. Today's C compilers simply discard the fractional part (the equivalent of rounding the quotient toward zero), making integer division results reliable across systems. For more information, see the C Standard Section 6.5.5.

C.7.4 Flexible Array Members

This section can be read after Section 10.3.

The *last* member of a struct may be an array of unspecified length, as in:

```
struct s {
   int arraySize;
   int array[];
};
```

This is called a flexible array member and is declared by specifying empty square brackets ([]) after the array's name. To allocate a struct with a flexible array member, use code such as:

```
int desiredSize = 5;
struct s *ptr;
ptr = malloc(sizeof(struct s) + sizeof(int) * desiredSize);
```

The sizeof operator ignores flexible array members, so sizeof(struct s) returns the size of all the struct's members *except* the flexible array member. The extra space we allocate with sizeof(int) * desiredSize is the flexible array's size.

Flexible-Array-Member Restrictions

There are many restrictions on flexible array members:

- A flexible array member may be declared only as a struct's last member, so each struct may contain at most one flexible array member.
- A flexible array cannot be a struct's only member—the struct must have one or more fixed members.
- A struct containing a flexible array may not be a member of another struct.
- A struct with a flexible array member must be allocated dynamically. You cannot fix the flexible array member's size at compile time.

For more information, see C99 Standard Section 6.7.2.1.

C.7.5 Type-Generic Math

[This section can be read after Section 5.3.]

C99 <tgmath.h> header provides type-generic macros for many math functions in <math.h>. For example, after including <tgmath.h> the expression sin(x) will call:

- sinf (the float version of sin) if x is a float,
- sin (which takes a double argument) if x is a double,
- sinl (the long double version of sin) if x is a long double, or
- one of csin, csinf or csinl (the sin functions for complex types) if x is a complex.

C11/C18 adds more generics capabilities, which we mention later in this appendix.

C.7.6 Inline Functions

[This section can be read after Section 5.5.]

You can declare inline functions by placing the keyword inline before the function declaration, as in:

inline void someFunction();

This can improve performance. Function calls take time. When we declare a function as inline, the program might no longer call that function. Instead, the compiler has the option to replace every call to an inline function with a copy of that function's code body. This improves the runtime performance, but it may increase the program's size. Declare functions as inline only if they are short and called frequently. If you change an inline function's definition, you must recompile any code that calls that function. The inline declaration is only advice to the compiler, which can decide to ignore it. Compilers also may optimize performance by inlining functions not declared inline. For more information, see C99 Standard Section 6.7.4.

C.7.7 __func__ Predefined Identifier

[This section can be read after Section 14.9.]

The __func__ predefined identifier is similar to the __FILE__ and __LINE__ preprocessor macros. When used in a function's body, __func__ is a string containing the

current function's name. Unlike __FILE__ and __LINE__, __func__ is a real variable, not a string literal visible at preprocessing time. So, __func__ cannot be concatenated with other literals during preprocessing.

C.7.8 va_copy Macro

[This section can be read after Section 15.2.]

Section 15.2 introduced the <stdarg.h> header and functions with variable-length argument lists. The va_copy macro takes two va_lists and copies its second argument into its first argument. This allows for multiple passes over a variable-length argument list without starting from the beginning each time.

C.8 CII/CI8 Features

C11/C18 refined and expanded C's capabilities. Some of C11/C18's features are considered optional. Microsoft Visual C++ provides only partial support for features that were added in C99 and C11/C18.

C.8.1 C11/C18 Headers

The following table lists the standard-library headers that were added in C11.

Header	Explanation
<stdalign.h></stdalign.h>	Provides type-alignment controls.
<stdatomic.h></stdatomic.h>	Provides uninterruptible access to objects used in multithreading.
<stdnoreturn.h></stdnoreturn.h>	Nonreturning functions.
<threads.h></threads.h>	Thread library (see Section C.9).
<uchar.h></uchar.h>	UTF-16 and UTF-32 character utilities.

C.8.2 quick_exit Function

In addition to exit (Section 15.5) and abort, C11/C18 provide function quick_exit (header <stdlib.h>) to terminate a program. Like exit, you call quick_exit and pass it an exit status as an argument—typically EXIT_SUCCESS or EXIT_FAILURE, but other platform-specific values are possible. The program returns the exit status value to the calling environment to indicate whether the program terminated successfully or an error occurred.

When called, quick_exit can, in turn, call up to at least 32 other functions to perform cleanup tasks. You register these functions, which must return void and have a void parameter list, with the at_quick_exit function (similar to atexit in Section 15.5). They're called in the reverse order from which they were registered. The motivation for functions quick_exit and at_quick_exit is explained at

http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1327.htm

C.8.3 Unicode® Support

Internationalization and localization is the process of creating software that supports multiple spoken languages and locale-specific requirements—such as displaying monetary formats. The Unicode® character set contains characters for many of the world's languages and symbols.

C11/C18 includes support for both the 16-bit (UTF-16) and 32-bit (UTF-32) Unicode character sets, making it easier for you to internationalize and localize your apps. Section 6.4.5 in the C18 standard discusses how to create Unicode string literals. Section 7.28 in the standard discusses the Unicode utilities header (<uchar.h>), which includes the new types char16_t and char32_t for UTF-16 and UTF-32 characters, respectively.

C.8.4 Noreturn Function Specifier

The _Noreturn function specifier indicates that a function will not return to its caller. For example, function exit (Section 15.5) terminates a program, so it does not return to its caller. Such C standard-library functions are now declared with _Noreturn. For example, the C11/C18 standards show function exit's prototype as:

```
_Noreturn void exit(int status);
```

If the compiler knows that a function does not return, it can perform various optimizations. It can also issue error messages if a _Noreturn function is inadvertently written to return.

C.8.5 Type-Generic Expressions

C11/C18's _Generic keyword provides a mechanism that you can use to create a macro (Chapter 14) that can invoke different type-specific versions of functions based on the macro's argument type. In C11/C18, _Generic is used to implement the features of the type-generic math header (<tgmath.h>). Many math functions provide separate versions that take float, double or long double arguments. In such cases, there is a macro that automatically invokes the corresponding type-specific version. For example, the macro ceil invokes the function ceilf when the argument is a float, ceil when the argument is a double and ceill when the argument is a long double. Section 6.5.1.1 of the C18 standard discusses the details of using _Generic.

C.8.6 Annex L: Analyzability and Undefined Behavior

The C11/C18 standards documents define the features of the language that compiler vendors must implement. Because of the extraordinary range of hardware and software platforms and other issues, the standard specifies in several places that the result of an operation is undefined behavior. These can raise security and reliability concerns. Whenever there's an undefined behavior, something happens that could leave a system open to attack or failure. The term "undefined behavior" appears approximately 50 times in the C18 standard document.

The people responsible for C11/C18's optional Annex L are from the CERT Division of Carnegie Mellon's Software Engineering Institute:

```
https://www.sei.cmu.edu/about/divisions/cert/index.cfm
```

They scrutinized all undefined behaviors mentioned in the C standard and discovered that these fall into two categories:

- those for which compiler implementers should be able to do something reasonable to avoid serious consequences—known as bounded undefined behaviors, and
- those for which implementers would not be able to do anything reasonable—known as *critical undefined behaviors*.

It turned out that most undefined behaviors belong to the first category. David Keaton (a researcher from the CERT Secure Coding Program) explains the categories in the following article:

```
https://insights.sei.cmu.edu/sei_blog/2012/06/improving-security-in-the-latest-c-programming-language-standard.html
```

The C11/C18 standard's Annex L identifies the critical undefined behaviors. Including this annex as part of the standard provides an opportunity for compiler implementors. A compiler that's Annex L compliant can be depended upon to do something reasonable for most of the undefined behaviors that might have been ignored in earlier implementations. Annex L still does not guarantee reasonable behavior for critical undefined behaviors. A program can determine whether the implementation is Annex L compliant by using conditional compilation directives (Section 14.5) that test whether the macro __STDC_ANALYZABLE__ is defined.

C.8.7 Memory Alignment Control

In Chapter 10, we discussed that computer platforms have different boundary alignment requirements, which could lead to struct objects requiring more memory than the total of their members' sizes. C11/C18 allows you to specify the boundary alignment requirements of any type using features of the <stdalign.h> header. _Alignas is used to specify alignment requirements. Operator alignof returns its argument's alignment requirement. Function aligned_alloc allows you to dynamically allocate memory for an object and specify its alignment requirements. For more details, see Section 6.2.8 of the C18 standard document.

C.8.8 Static Assertions

In Section 14.10, you learned that C's assert macro tests an expression's value at execution time. If the condition's value is false, assert prints an error message and calls function abort to terminate the program. This is useful for debugging purposes. C11/C18 provides _Static_assert for compile-time assertions that test constant expressions after the preprocessor executes and at a point during compilation when the expressions' types are known. For more details, see Section 6.7.10 of the C18 standard document.

C.8.9 Floating-Point Types

C11/C18 compilers may optionally provide support for the IEC 60559 floating-point arithmetic standard. Among its features, IEC 60559 defines how floating-point arithmetic should be performed to ensure that you always get the same results across implementations, whether the calculations are performed by hardware, software or both. You can learn more about this standard at:

http://www.iso.org/iso/iso_catalogue/catalogue_tc/
 catalogue_detail.htm?csnumber=57469

C.9 Case Study: Performance with Multithreading and Multicore Systems

It would be nice if we could focus our attention on performing only one task at a time and doing it well. That's usually difficult to do in a complex world where there's so much going on at once. This section presents C's capabilities for creating and managing multiple tasks. As we'll demonstrate, this can significantly improve program performance and responsiveness.

Concurrency vs. Parallelism

When we say that two tasks are operating concurrently, we mean that they're both making progress at once. Until the early 2000s, most computers had only a single processor. Operating systems on such computers execute tasks concurrently by rapidly switching between them, doing a small portion of each before moving on to the next so that all tasks keep progressing. For example, it's common for your computer to perform many tasks concurrently, such as compiling a program, sending a file to a printer, receiving e-mail messages, posting a tweet, uploading a video to YouTube, uploading a photo to Facebook or Instagram and more.



When we say that two tasks are operating in parallel, we mean that they're executing truly simultaneously. In this sense, parallelism is a subset of concurrency. The human body performs a great variety of operations in parallel. For example, respiration, blood circulation, digestion, thinking and walking can occur in parallel, as can all the senses—sight, hearing, touch, smell and taste.

No one knows exactly how powerful the human brain is, but various articles state that it has the equivalent of 100 billion "processors"^{3,4,5} and one article we found says the brain has the equivalent of "five million contemporary 200 million transistor chip

^{3. &}quot;How Many Supercomputers Would Fit Inside Your Brain?" Accessed December 4, 2020. https://fountainmagazine.com/2016/issue-111-may-june-2016/how-many-supercomputers-would-fit-inside-your-brain.

^{4. &}quot;When compared to a computer CPU, is human brain single-core or multi-core?" Accessed December 4, 2020. https://www.quora.com/When-compared-to-a-computer-CPU-is-human-brain-single-core-or-multi-core/answer/Frank-Heile.

^{5. &}quot;Which is the equivalent processing of human brain in terms of computer processing?" Accessed December 4, 2020. https://cs.stackexchange.com/questions/20016/which-is-the-equivalent-processing-of-human-brain-in-terms-of-computer-processin/40075.

cores."6 Today's multicore computers have multiple processors that can perform tasks in parallel.

C Concurrency

C programs can have multiple threads of execution, each with its own function-call stack and program counter (which keeps track of the next instruction to execute), allowing that thread to execute concurrently with other threads. This capability is called multithreading.

A problem with single-threaded applications is that lengthy activities must complete before others can begin—that can lead to poor responsiveness. In a multithreaded application, threads can be distributed across multiple available cores so that multiple tasks execute in parallel, enabling the application to operate more efficiently. Multithreading can also increase performance on single-processor systems—when one thread cannot proceed (because, for example, it's waiting for an event to occur, such as a timer expiration or the completion of an I/O operation), another can use the processor.

A single-core system with multithreading can have several threads executing concurrently, but not in parallel. A multicore system with multithreading can have some threads executing concurrently and some executing truly in parallel.

Multicore Systems

Though multithreading has been around since the late 1960s,⁷ interest in it is rising quickly due to the proliferation of multicore systems. Smartphones and tablets commonly contain multicore processors.

The first multicore CPU was introduced by IBM in 2001.8 Most new processors today have at least two cores, with three, four and eight cores now common. Apple's recently introduced M1 processor has eight CPU cores and up to eight additional graphics processing unit (GPU) cores. 9 AMD has desktop processors with up to 32 cores. 10 Intel has processors with up to 18 cores 11 for consumers and high-end processors with up to 72 cores for supercomputers, high-end servers and ultra-high-performance desktop systems.¹² To take full advantage of multicore architecture, you need to write multithreaded applications.



^{6. &}quot;Neural waves of brain." December 4, 2020. https://biophilic.blogspot.com/2011/05/ neural-waves-of-brain.html.

^{7. &}quot;Thread (computing)" Accessed December 4, 2020. https://en.wikipedia.org/wiki/ Thread_(computing).

^{8. &}quot;Power 4: The First Multi-Core, 1GHz Processor" Accessed December 4, 2020. https:// www.ibm.com/ibm/history/ibm100/us/en/icons/power4/.

^{9. &}quot;Apple unleashes M1." Accessed November 18, 2020. https://www.apple.com/newsroom/ 2020/11/apple-unleashes-m1.

^{10. &}quot;AMD unveils world's most powerful desktop CPUs." Accessed November 18, 2020. https:/ /www.zdnet.com/article/amd-unveils-worlds-most-powerful-desktop-cpus/.

^{11. &}quot;Intel Core Processor Family." Accessed November 18, 2020. https://www.intel.com/ content/www/us/en/products/processors/core.html.

^{12. &}quot;Xeon Phi" Accessed November 18, 2020. https://en.wikipedia.org/wiki/Xeon_Phi.

Concurrent Programming Is Difficult

Writing multithreaded programs can be tricky. Although the human mind can perform functions concurrently, people find it difficult to jump between parallel trains of thought. To see why multithreaded programs can be challenging to write and understand, try the following experiment: Open three books to page 1 and try reading the books concurrently. Read a few words from the first book, then a few from the second, then a few from the third, then loop back and read the next few words from the first book, and so on. After this experiment, you'll appreciate many of multithreading's challenges. You must

- *switch* between the books,
- read briefly,
- remember your place in each book,
- move the book you're reading closer so that you can see it and
- push the books you're not reading aside.

And, amid all this chaos of rapidly repeating these tasks, you must try to *comprehend* the content of the books!

Standard Multithreading Implementation

Previously, C multithreading libraries were nonstandard, platform-specific language extensions. C programmers often want their code to be portable across platforms—this is a key benefit of standardized multithreading. The <threads.h> header declares the (optional) multithreading capabilities for writing more portable multithreaded code.

Microsoft Visual C++ and Apple's version of the Clang compiler in Xcode do not support <threads.h>. So, we tested this section's examples, using:

- GNU gcc 10.2 on Ubuntu Linux,
- GNU gcc 10.2 in the GNU Compiler Collection Docker container, which can run on Windows, macOS and Linux,
- GNU gcc 10.2 on Ubuntu Linux running in the Windows Subsystem for Linux (WSL), and
- Clang 10.0 on Linux.

In this section, we introduce basic features that enable you to create and execute threads and simple multithreaded applications. At the end of the section, we mention several other multithreading features you'll want to explore if you would like to create more sophisticated multithreaded applications.

Running Multithreaded Programs

When you run a program, its tasks compete for the processors' attention with

- the operating system,
- other programs, and
- other activities the operating system runs on your behalf.

When you execute the examples in this section, the time to perform each calculation will vary based on your computer's

- processor speed,
- number of processor cores, and
- what's running on your computer.

It's like driving to a store—the time it takes can vary, based on traffic conditions, weather and other factors. Some days the drive might take 10 minutes, but it could take longer during rush hour or bad weather.

There's also *overhead* inherent in multithreading itself. Simply dividing a task into PERF two threads and running it on a dual-core system does *not* run it twice as fast, though it will typically run faster than performing the thread's tasks in sequence. Executing a multithreaded application on a single-core processor can actually take longer than simply performing the thread's tasks in sequence.

Overview of This Section's Examples

To provide a convincing demonstration of multithreading's power on a multicore system, this section presents two programs:

- One performs two compute-intensive calculations *sequentially*.
- The other executes the same compute-intensive calculations in *parallel threads*.

The outputs shown were produced using the GNU Compiler Collection Docker container. Docker allows you to specify the number of cores dedicated to the container when you launch it by using the command-line argument:

```
--cpus=numberOfCores
```

We executed each program using the Docker container with one core then with two to show the programs' performance in each scenario. We show the individual calculation times and total calculation time for each program. The outputs show the time improvement when the multithreaded program executes on two cores instead of just one.

C.9.1 Example: Sequential Execution of Two Compute-Intensive Tasks

Lines 35–42 of Fig. C.5 define the recursive fibonacci function, originally discussed in Section 5.15. As we saw in that section, for larger Fibonacci values, the recursive implementation can require significant computation time. This example sequentially performs the calculations fibonacci (50) (line 14) and fibonacci (49) (line 23).

```
1 // figC_05.c
2 // Fibonacci calculations performed sequentially
3 #include <stdio.h>
4 #include <time.h>
```

Fig. C.5 | Fibonacci calculations performed sequentially. (Part 1 of 3.)

```
5
    long long int fibonacci(int n); // function prototype
 7
    int main(void) {
 8
 9
       puts("Sequential calls to fibonacci(50) and fibonacci(49)");
10
       // calculate fibonacci value for 50
П
       time_t startTime1 = time(NULL);
12
       puts("Calculating fibonacci(50)");
13
       long long int result1 = fibonacci(50);
14
       time_t endTime1 = time(NULL);
15
16
       printf("fibonacci(50) = %llu\n", result1);
17
       printf("Calculation time = %f minutes\n\n",
18
          difftime(endTime1, startTime1) / 60.0);
19
20
21
       time_t startTime2 = time(NULL);
22
       puts("Calculating fibonacci(49)");
23
       long long int result2 = fibonacci(49);
24
       time_t endTime2 = time(NULL);
25
       printf("fibonacci(49) = %1lu\n", result2);
26
27
       printf("Calculation time = %f minutes\n\n",
          difftime(endTime2, startTime2) / 60.0);
28
29
30
       printf("Total calculation time = %f minutes\n",
31
          difftime(endTime2, startTime1) / 60.0);
32
    }
33
34
    // Recursively calculates fibonacci numbers
    long long int fibonacci(int n) {
35
36
       if (0 == n \mid | 1 == n) \{ // base case \}
37
          return n;
       }
38
       else { // recursive step
39
          return fibonacci(n - 1) + fibonacci(n - 2);
40
41
42
   }
```

a) Run on a Docker Container with One Core

```
Sequential calls to fibonacci(50) and fibonacci(49)
Calculating fibonacci(50)
fibonacci(50) = 12586269025
Calculation time = 1.700000 minutes

Calculating fibonacci(49)
fibonacci(49) = 7778742049
Calculation time = 1.050000 minutes

Total calculation time = 2.750000 minutes
```

Fig. C.5 | Fibonacci calculations performed sequentially. (Part 2 of 3.)

b) Run on a Docker Container with Two Cores

```
Sequential calls to fibonacci(50) and fibonacci(49)
Calculating fibonacci(50)
fibonacci(50) = 12586269025
Calculation time = 1.666667 minutes

Calculating fibonacci(49)
fibonacci(49) = 7778742049
Calculation time = 1.066667 minutes

Total calculation time = 2.733333 minutes
```

Fig. C.5 | Fibonacci calculations performed sequentially. (Part 3 of 3.)

Before and after each fibonacci call, we capture the time so we can determine the calculation's total processing time. We also use these times to calculate the total time required for both calculations. Lines 19, 28 and 31 use function difftime (from header <time.h>) to determine the number of seconds between two times.

The first output shows the program's results in the GNU Compiler Docker Container using one core. The second shows the results of running the program with the Docker container configured to use two cores. Figure C.5 does not use multithreading, so the program can execute only on one core, even on the two-core Docker container. In our testing, running the programs multiple times with one and two cores produced slightly different results each time. Using a single core generally took longer because the processor was being shared between this program and Docker.

C.9.2 Example: Multithreaded Execution of Two Compute-Intensive Tasks

Figure C.6 also uses the recursive fibonacci function but executes each call in a *separate thread*. To compile this program with GNU gcc—either in Linux or in the GNU Compiler Collection Docker container—use the command:

```
gcc -std=c18 figC_06.c -pthread
```

The linker uses the -pthread option to link our program to the Linux operating system's threading library. If you have Clang on Linux, you can compile the program with:

```
clang -std=c18 figC_06.c -pthread
```

```
1  // figC_06.c
2  // Fibonacci calculations performed in separate threads
3  #include <stdio.h>
4  #include <threads.h>
5  #include <time.h>
6
7  #define NUMBER_OF_THREADS 2
```

Fig. C.6 | Fibonacci calculations performed in separate threads. (Part 1 of 4.)

```
8
 9
    int startFibonacci(void *nPtr);
   long long int fibonacci(int n);
10
П
   typedef struct ThreadData {
12
       time_t startTime; // time thread starts processing
13
       time_t endTime; // time thread finishes processing
14
15
       int number; // fibonacci number to calculate
    } ThreadData; // end struct ThreadData
16
17
    int main(void) {
18
       // data passed to the threads; uses designated initializers
19
       ThreadData data[NUMBER_OF_THREADS] =
20
21
          \{[0] = \{.number = 50\},\
22
           [1] = \{.number = 49\}\};
23
24
       // each thread needs a thread identifier of type thrd_t
25
       thrd_t threads[NUMBER_OF_THREADS];
26
27
       puts("fibonacci(50) and fibonacci(49) in separate threads");
28
29
       // create and start the threads
30
       for (size_t i = 0; i < NUMBER_OF_THREADS; ++i) {</pre>
          printf("Starting thread to calculate fibonacci(%d)\n",
31
             data[i].number);
32
33
34
          // create a thread and check whether creation was successful
35
          if (thrd_create(&threads[i], startFibonacci, &data[i]) !=
             thrd_success) {
36
37
             puts("Failed to create thread");
38
39
       }
40
41
       // wait for each of the calculations to complete
42
       for (size_t i = 0; i < NUMBER_OF_THREADS; ++i) {</pre>
43
          thrd_join(threads[i], NULL);
44
45
       // determine time that first thread started
46
       time_t startTime = (data[0].startTime < data[1].startTime) ?</pre>
47
          data[0].startTime : data[1].startTime;
48
49
       // determine time that last thread terminated
50
51
       time_t endTime = (data[0].endTime > data[1].endTime) ?
52
          data[0].endTime : data[1].endTime;
53
54
       // display total time for calculations
55
       printf("Total calculation time = %f minutes\n",
56
          difftime(endTime, startTime) / 60.0);
57 }
58
```

Fig. C.6 | Fibonacci calculations performed in separate threads. (Part 2 of 4.)

```
59
    // Called by a thread to begin recursive Fibonacci calculation
    int startFibonacci(void *ptr) {
60
61
       // cast ptr to ThreadData * so we can access arguments
62
       ThreadData *dataPtr = (ThreadData *) ptr;
63
64
       dataPtr->startTime = time(NULL); // time before calculation
65
       printf("Calculating fibonacci(%d)\n", dataPtr->number);
66
       printf("fibonacci(%d) = %11d\n",
67
          dataPtr->number, fibonacci(dataPtr->number));
68
69
70
       dataPtr->endTime = time(NULL); // time after calculation
71
72
       printf("Calculation time = %f minutes\n\n",
73
          difftime(dataPtr->endTime, dataPtr->startTime) / 60.0);
74
       return thrd_success;
75
    }
76
77
    // Recursively calculates fibonacci numbers
    long long int fibonacci(int n) {
78
79
       if (0 == n \mid | 1 == n) \{ // base case \}
80
          return n;
       }
81
82
       else { // recursive step
          return fibonacci(n - 1) + fibonacci(n - 2);
83
84
       }
85 }
a) Run on a Docker Container with Two Cores
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci (50)
Starting thread to calculate fibonacci (49)
Calculating fibonacci(50)
Calculating fibonacci (49)
fibonacci(49) = 7778742049
Calculation time = 1.083333 minutes
fibonacci(50) = 12586269025
Calculation time = 1.733333 minutes
```

b) Run on a Docker Container with Two Cores

Total calculation time = 1.733333 minutes

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci(50)
Starting thread to calculate fibonacci(49)
Calculating fibonacci(50)
Calculating fibonacci(49)
fibonacci(49) = 7778742049
Calculation time = 1.033333 minutes
```

Fig. C.6 | Fibonacci calculations performed in separate threads. (Part 3 of 4.)

```
fibonacci(50) = 12586269025
Calculation time = 1.600000 minutes
Total calculation time = 1.600000 minutes
```

c) Run on a Docker Container with One Core

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci(50)
Starting thread to calculate fibonacci(49)
Calculating fibonacci(50)
Calculating fibonacci(49)
fibonacci(49) = 7778742049
Calculation time = 2.150000 minutes

fibonacci(50) = 12586269025
Calculation time = 2.816667 minutes

Total calculation time = 2.816667 minutes
```

d) Run on a Docker Container with One Core

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci(50)
Starting thread to calculate fibonacci(49)
Calculating fibonacci(50)
Calculating fibonacci(49)
fibonacci(49) = 7778742049
Calculation time = 2.166667 minutes

fibonacci(50) = 12586269025
Calculation time = 2.833333 minutes

Total calculation time = 2.833333 minutes
```

Fig. C.6 | Fibonacci calculations performed in separate threads. (Part 4 of 4.)

The first two outputs show the multithreaded Fibonacci example executing on a two-core Docker container. Though execution times varied, the total time to perform both Fibonacci calculations (in our tests) was less than Fig. C.5's sequential executions—the total execution time was the same as the longer fibonacci (50) calculation. Splitting our program into two threads enabled the two Fibonacci calculations to execute simultaneously—one on each core. The last two outputs show the example executing on a one-core Docker container. Again, times varied for each execution, but the total time was *more* than Fig. C.5's sequential executions due to the overhead of sharing *one* processor among the program's threads and Docker.

struct ThreadData

Lines 12–16 define a ThreadData struct type containing the number we pass to function fibonacci and two time_t members where we store the time before and after

each thread's fibonacci call. The function that each thread executes in this example receives a ThreadData object as its argument. Lines 20–22 create a ThreadData array and use designated initializers (introduced in Section C.3) to set their number members to 50 and 49—the Fibonacci numbers we'll calculate.

thrd t

Line 25 creates an array of **thrd_t objects**. When you create a thread, the multi-threading library creates a unique *thread ID* and stores it in a thrd_t object. The thread's ID can be used with various multithreading functions.

Creating and Executing a Thread

Lines 30–39 create two threads by calling function **thrd_create** (line 35). The function's three arguments are:

- A thrd_t pointer that thrd_create uses to store the thread's ID.
- A pointer to a function (startFibonacci) specifying the task to perform in the thread—This function must return an int and receive a void * pointer representing the function's argument. The int return value represents the thread's state when it terminates. The void * pointer enables this function to receive an argument of any type that's appropriate for your application—in our case, a pointer to a ThreadData object. Recall that any pointer type can be assigned to a void *.
- A void * pointer to the argument that thrd_create will pass to the function in the second argument.

Function thrd_create returns thrd_success if the thread is created, thrd_nomem if there was not enough memory to allocate the thread or thrd_error otherwise. If the thread is created successfully, the function specified as thrd_create's second argument begins executing in the new thread.

Joining the Threads

To ensure that the program does not terminate until the threads terminate, lines 42–44 call **thrd_join** for each thread. This causes the program to *wait* until both threads terminate before executing the remaining code in main. Function thrd_join receives the thrd_t ID of the thread to join and an int pointer where thrd_join stores the status the thread returns when it terminates—if you don't need this status, pass NULL for this argument.

Calculating the Execution Times

After the threads terminate, lines 47–56 calculate and display the total execution time by determining the time difference between the time the first thread started and the second thread ended.

Function startFibonacci

Function startFibonacci (lines 60-75) specifies the tasks to perform. In this case, we:

746 Appendix C Multithreading/Multicore and Other C18/C11/C99 Topics

- call fibonacci to perform a calculation recursively,
- time the calculation,
- display the calculation's result, and
- display the time the calculation took (as we did in Fig. C.5).

The thread executes until startFibonacci returns the thread's status (thrd_success, line 74), at which point the thread terminates. When this function finishes executing, its corresponding thread terminates.

C.9.3 Other Multithreading Features

There are many other multithreading features, including _Atomic variables and atomic operations, thread-local storage, conditions and mutexes. For more information on these topics, see the C18 Standard Sections 6.7.2.4, 6.7.3, 7.17 and 7.26 and the following blog post and article:

 $\label{local-comblog} $$ $$ https://smartbear.com/blog/test-and-monitor/c11-a-new-c-standard-aiming-at-safer-programming/$

http://lwn.net/Articles/508220/

For documentation, see the threads page at:

https://en.cppreference.com/w/c/thread



Intro to Object-Oriented Programming Concepts

D.1 Introduction

After you learn C, you'll likely learn one or more C-based or C-influenced object-oriented languages. These include Java, C++, C#, Objective-C, Python, Swift and many more. These languages often support several programming paradigms:

- procedural programming,
- object-oriented programming,
- generic programming, and
- functional-style programming.

This appendix presents a friendly overview of object-oriented programming terminology and concepts.

D.2 Object-Oriented Programming Languages

C spawned a whole new generation of programming languages that go beyond C's procedural-programming model. As demands for new and more powerful software soar, building software quickly, correctly and economically is important. Objects, or more precisely, the classes objects come from, are essentially reusable software components. There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc. Almost any noun can be reasonably represented as a software object in terms of attributes (e.g., name, color and size) and behaviors (e.g., calculating, moving and communicating). Software-development groups can use a modular, object-oriented design-and-implementation approach to be more productive than with earlier popular techniques. Object-oriented programs are often easier to understand, correct and modify.

D.3 Automobile as an Object¹

To help you understand objects and their contents, consider a simple analogy. Suppose you want to drive a car and make it go faster by pressing its accelerator pedal. What must happen before you can do this? Well, before you can drive a car, someone has to design it. A car typically begins as engineering drawings, similar to the blue-prints that describe a house's design. These drawings include the design for an accelerator pedal. The pedal hides from the driver the complex mechanisms that make the car go faster, just as the brake pedal "hides" the mechanisms that slow the car, and the steering wheel "hides" the mechanisms that turn the car. This enables people to drive a car even if they have little to no knowledge of how engines, braking and steering mechanisms work.

Just as you cannot cook meals in a kitchen's blueprint, you cannot drive a car's engineering drawings. Before you can drive a car, it must be built from the engineering drawings that describe it. A completed car has an actual accelerator pedal to make it go faster, but even that's not enough. The car won't accelerate on its own (hopefully!), so the driver must press the pedal to accelerate the car.

D.4 Methods and Classes

Performing a task in an object-oriented program requires a method. Methods house the program statements that perform their tasks. Each method hides these statements from its user, just as a car's accelerator pedal hides from the driver the mechanisms that make the car go faster. In object-oriented programming, a program unit called a class houses the set of methods that perform the class's tasks. For example, a class representing a bank account might contain one method to deposit money into an account, another to withdraw money from an account and a third to inquire what the account's balance is. A class is similar in concept to a car's engineering drawings, which house the designs for the accelerator pedal, steering wheel, and so on.

D.5 Instantiation

Just as someone has to build a car from its engineering drawings before you can drive a car, you must build an object of a class before a program can perform the tasks that the class's methods define. The process of doing this is called instantiation. An object is then referred to as an instance of its class.

D.6 Reuse

Just as a car's engineering drawings can be reused many times to build many cars, you can reuse a class many times to build many objects. Reusing existing classes when building new classes and programs saves time and effort. Reuse also helps you build

^{1.} As you read the remainder of this appendix, think of how self-driving cars would affect the discussion.

more reliable and effective systems. Existing classes and components often have undergone extensive testing and debugging (finding and removing errors) and performance tuning. Just as the notion of interchangeable parts was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that has been spurred by object technology.

In object-oriented languages like C++, Java, C#, Python, Swift and many more, you'll typically use a **building-block approach** to create your programs. To avoid reinventing the wheel, you'll use existing high-quality pieces wherever possible. This software reuse is a key benefit of object-oriented programming.

D.7 Messages and Method Calls

When you drive a car, pressing its gas pedal sends a message to the car to perform a task—that is, to go faster. Similarly, you send messages to an object. Each message is implemented as a **method call** that tells a method of the object to perform its task. For example, a program might call a bank-account object's deposit method to increase the account's balance by a specified amount.

D.8 Attributes and Instance Variables

A car, besides having capabilities to accomplish tasks, also has attributes, such as its color, its number of doors, the amount of gas in its tank, its current speed and its record of total miles driven (that is, its odometer reading). Like its capabilities, the car's attributes are represented as part of its design in its engineering diagrams (which, for example, include an odometer and a fuel gauge). As you drive an actual car, these attributes are carried along with the car. Every car maintains its own attributes. For example, each car knows how much gas is in its own gas tank, but not how much is in other cars' tanks.

An object, similarly, has attributes that it carries along as it's used in a program. These attributes are specified as part of the object's class. For example, a bank-account object has a balance attribute representing the amount of money in the account. Each bank-account object knows the balance in the account it represents, but not the balances of the bank's other accounts. Attributes are specified by the class's instance variables. A class's (and its objects') attributes and methods are intimately related, so classes wrap together their attributes and methods.

D.9 Inheritance

A new class of objects can be created conveniently by inheritance—the new class (called the subclass) starts with the characteristics of an existing class (called the superclass), possibly customizing them and adding unique characteristics of its own. In our car analogy, an object of class convertible certainly is an object of the more general class "automobile," but more specifically, the roof can be raised or lowered.

D.10 Object-Oriented Analysis and Design (OOAD)

Many programmers create the code (i.e., the program instructions) for their programs without an initial planning phase. This approach may work for small programs like those we presented in the early chapters of this book. But what if you were asked to create a software system to control thousands of automated teller machines for a bank? Or suppose you were asked to work on a team of 1,000 software developers building the next generation of the U.S. air traffic control system?

To create the best solutions for projects so large and complex, you should follow a detailed analysis process for determining your project's requirements—that is, define what the system is supposed to do. You'd then develop a design that satisfies those requirements—that is, specify how the system should do it. Ideally, before writing any code, you'd go through this process and carefully review the design and have your design reviewed by other software professionals. If this process involves analyzing and designing your system from an object-oriented perspective, it's called an object-oriented analysis-and-design (OOAD) process. Programming in an object-oriented language is called object-oriented programming (OOP) and allows you to implement an object-oriented design as a working system.